SELF-RECONFIGURABLE MULTI-ROBOT SYSTEMS

A Dissertation
Presented to
The Academic Faculty

by

Daniel Pickem

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Robotics

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2016

SELF-RECONFIGURABLE MULTI-ROBOT SYSTEMS

Approved by:

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

Professor Jeff S. Shamma, Co-Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Justin Romberg
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Martha A. Grover
School of Chemical and Biological Engineering
Georgia Institute of Technology

Professor Jun Ueda
School of Mechanical Engineering
Georgia Institute of Technology

Date Approved: 28 March 2016

ACKNOWLEDGEMENTS

First and foremost I would like to thank both my advisors, Dr. Magnus Egerstedt and Dr. Jeff S. Shamma. I was fortunate enough to work with not one, but two great minds during my graduate life and benefit from their vast knowledge in control theory, game theory, and robotics. Their two very different advising styles enriched my Ph.D. experience through both fast-paced, high-intensity meetings as well as extended meetings focused on discussion, learning, and brainstorming. What both of them share, however, are their unmatched abilities as mentors and teachers. Not only are their lectures inspiring and entertaining, but their guidance in both my academic and professional life has been invaluable. My interactions with Jeff and Magnus have not only shaped my way of thinking and approaching problems but my approach to research as a whole.

I would also like to thank my committee members for their valuable feedback and suggestions that guided the latter stages of my research. Their diverse perspectives and insights enabled interesting extensions to my research.

Countless interactions and discussions with members of the GRITS lab and the DCL lab were instrumental in charting the waters of my Ph.D. and furthering my research agenda. I would not want to miss the opportunity to thank each and every one of them for their input, ideas, and feedback on my doctoral research over the years.

I would also like to express my sincere gratitude to my parents and sister - Eva, Herbert and Judith - for their love and support over the many years of my student life - both in Austria and even more so in the US. They always encouraged me to seek out new heights in my education, even if that meant watching me move farther and farther away. No matter the large distances, they were always there for me. From a young age, they not only instilled a passion for creating and building in me but also an insatiable curiosity. They never stopped me when I disassembled or accidentally blew up yet another piece of electronics just to learn about its inner workings. I owe my adventuresomeness and curiosity to them! Liebe Mutti,

lieber Papa, liebe Jutzi, besten Dank für eure Unterstützung und Liebe über all die Jahre. Es bedeutet die Welt für mich! Of course, I would also like to thank Yulia's family for their sincere interest, kind wishes, and encouragement over the years.

Finally, I cannot even begin to express how thankful I am for the unconditional love, unwaivering support, and true friendship of my significant other - Yulia. I could always rely on her for advise, counsel, and encouragement. Her attentiveness, eye for detail, and critical thinking helped me improve my work countless times and not only shaped my dissertation but my research as a whole. Yulia always had an open ear for when I started talking about Markov chains or robots yet another time and her challenging questions routinely made me rethink aspects of my research. Throughout my years as a graduate student, Yulia was always there for me, supporting me emotionally and being my source of inspiration. She made me grow as a researcher, but more importantly as a person, friend, and partner. Yulia is not only my strongest pillar of support but the most loving woman in my life. Without her, I would not be where and who I am today. Я очень сильно тебя люблю, моя дорогая Юленька!

Contents

List of Tables

List of Figures

SUMMARY

Self-reconfigurable robotic systems are variable-morphology machines capable of changing their overall structure by rearranging the modules they are composed of. Individual modules are capable of connecting and disconnecting to and from one another, which allows the robot to adapt to changing environments. Optimally reconfiguring such systems is computationally prohibitive and thus in general self-reconfiguration approaches aim at approximating optimal solutions. Nonetheless, even for approximate solutions, centralized methods scale poorly in the number of modules. Therefore, the objective of this research is the development of decentralized self-reconfiguration methods for modular robotic systems.

Building on completeness results of the centralized algorithms in this work, decentralized methods are developed that guarantee convergence to a given target shape. A game-theoretic approach lays the theoretical foundation of a novel potential game-based formulation of the self-reconfiguration problem. Stochastic convergence guarantees are provided for a large class of utility functions used by purely self-interested agents. Furthermore, two extensions to the basic game-theoretic learning algorithm are proposed that enable agents to modify the algorithms' parameters during runtime and improve convergence times. The flexibility in the choice of utility functions together with runtime adaptability makes the presented approach and the underlying theory suitable for a range of problems that rely on decentralized local control to guarantee global, emerging properties.

The experimental evaluation of the presented algorithms relies on a newly developed multi-robotic testbed called the "Robotarium" that is equipped with novel custom-designed miniature robots, the "GRITSBots". The Robotarium provides hardware validation of self-reconfiguration on robots but more importantly introduces a novel paradigm for remote accessibility of multi-agent testbeds with the goal of lowering the barrier to entrance into the field of multi-robot research and education.

Chapter I

INTRODUCTION

Self-reconfigurable systems are comprised of individual modules which are able to connect to and disconnect from one another to form larger structures. These systems therefore have the ability to change their morphology, structure, and functionality through changing the relative positions of their modules. Changes to the system's configuration allow the aggregate robot to adapt to new tasks, changing environments, or replace broken or malfunctioning modules. In the context of search and rescue, for example, such a change in morphology could be warranted by a change of terrain from flat surfaces to rough terrain. A modular robot could then reconfigure from a wheeled configuration into one that features legged locomotion and subsequently traverse the challenging terrain. Another benefit of modular robots is that broken modules can be replaced by functional ones or new modules can be added without changing the general functionality of the structure [196, 71, 164].

Their modular architecture allows fine-grained control of self-reconfigurable systems (SRS) but introduces significant computational and controls-related complexities as the number of modules is scaled up and the number of degrees of freedom increases. On the one hand, as mentioned in [1], the number of possible configurations grows exponentially in both the number of modules and the number of connectors per module. On the other hand, the self-reconfiguration problem (SRP) itself is NP-hard due to the highly combinatorial nature of the problem, which is stated as follows. Solving the self-reconfiguration problem requires planning a sequence of individual module motions that optimally transforms an initial into a desired target configuration (an example of such a sequence is shown in Fig. 1).[1] Because of the intractability of computing optimal solutions for large self-reconfigurable systems, the majority of approaches is either limited to small systems or aims at approximating optimal

[1] Optimality for the SRP can be formulated in a number of ways but is most often stated as the minimum cumulative distance traveled by all modules.

solutions.

The overarching goal of the presented work is therefore the development of methods that make the control of large modular robotic systems tractable by sacrificing optimality in favor of scalability. In particular, the objective of this research is twofold. On the one hand, it explores scalable distributed algorithms for the control of self-reconfigurable systems in Chapter 3 - 5. On the other hand, a full-fledged multi-robot testbed based on custom-designed miniature robots is developed in Chapter 6 that allows the experimental verification of the presented algorithms as well as multi-agent algorithms in general. The algorithmic and theoretical components of this work place the focus on scalable and decentralized approaches that solve the self-reconfiguration problem in a provably complete manner and thus guarantee convergence to the desired target configuration. Novel approaches for the decentralized reconfiguration of modular robots from arbitrary initial configurations to desired target configurations are presented in Chapter 4 and Chapter 5.

Before these decentralized methods are developed, Chapter 3 explores centralized planning methods and aims at establishing an understanding of the challenges associated with self-reconfiguration. More specifically, Chapter 3 investigates the difficulties of centralized planning in such high-dimensional spaces and presents novel algorithms that can be applied to homogeneous as well as heterogeneous systems. While heterogeneous systems feature modules that differ in one or more properties - for example shape, size, or capabilities - homogeneous systems are comprised of completely identical and interchangeable modules. This module interchangeability in homogeneous systems typically simplifies the reconfiguration planning problem, but just as for heterogeneous systems, the potential for deadlocks exist. Therefore, the focus of Chapter 3 lies in the detection and avoidance of deadlocks such that the assembly of the target configuration can be guaranteed. In fact, the main contribution of Chapter 3 is the development of provably complete algorithms for both homogeneous and heterogeneous self-reconfiguration planning. However, given the high computational complexity of the self-reconfiguration problem (even after the requirement of optimality is dropped), centralized methods scale poorly in the number of modules. Therefore, the methods shown in Chapter 3 aim at establishing completeness in the algorithmic sense rather than

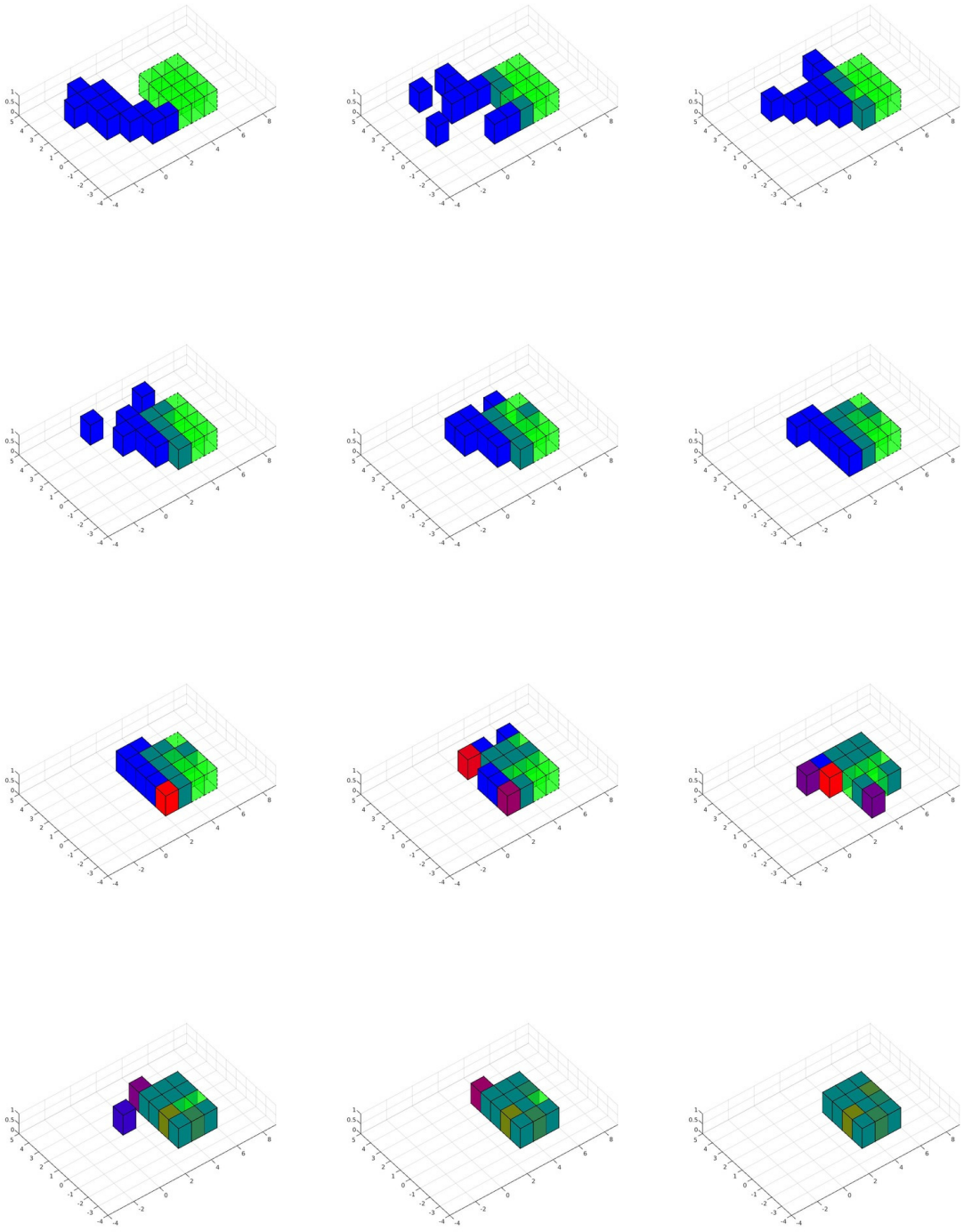


Figure 1: Example of a self-reconfiguration sequence from an initial random 2D configuration (left, opaque blue) to a 2D target configuration (right, semi-transparent green) using the adaptive learning rate strategy outlined in Chapter 5 (also see [147] and [148]). Note that not every time step is shown.

solving large problem instances. These completeness results lay the theoretical groundwork for the decentralized methods in Chapter 4 and Chapter 5, which are the main contributions of this work.

The literature discusses numerous centralized and decentralized solutions methods for the SRP. However, the majority of algorithms include caveats that potentially compromise their scalability or generality. For example, some distributed methods require a large amount of communication [61] or precomputation [60, 145], others either focus on locomotion [30] or functional target shape assemblies alone [109]. Distributed approaches have often relied on precomputation of rulesets [66, 145], policies [60], or entire sets of paths/folding schemata of agents [42]. Chapter 4 aims at rectifying these shortcomings by presenting a fully decentralized approach to homogeneous self-reconfiguration for which no precomputation is required. Our method guarantees convergence to the target configuration even though each module acts as a purely self-interested decision maker with local information only (and therefore little communication overhead). Modules are represented by game-theoretic agents and the overall self-reconfiguration problem is formulated as a potential game that can be solved using game-theoretic tools. In particular, the main contribution of this section is a novel game-theoretic learning algorithm that guarantees stochastic convergence to the target configuration for a large class of utility functions.[2] Flexibility in the choice of utility functions makes the presented approach and the underlying theory suitable for a wide array of problems that rely on decentralized local control to guarantee globally emerging properties.

Future applications for self-assembling and self-reconfiguring systems are plentiful and widely varied. On the more practical end of the spectrum, these applications include reconfigurable structures such as furniture, vehicles, or buildings, planetary exploration, or adaptable search and rescue robots. More futuristic applications range from self-assembling nano-robots (for example for health care), programmable matter (for example to change optical or acoustical properties of materials), or even accelerated robotic evolution (see [131]).

[2]Utility functions incentivize agents on a local level in such a way as to guarantee global properties, for example, the assembly of the target configuration.

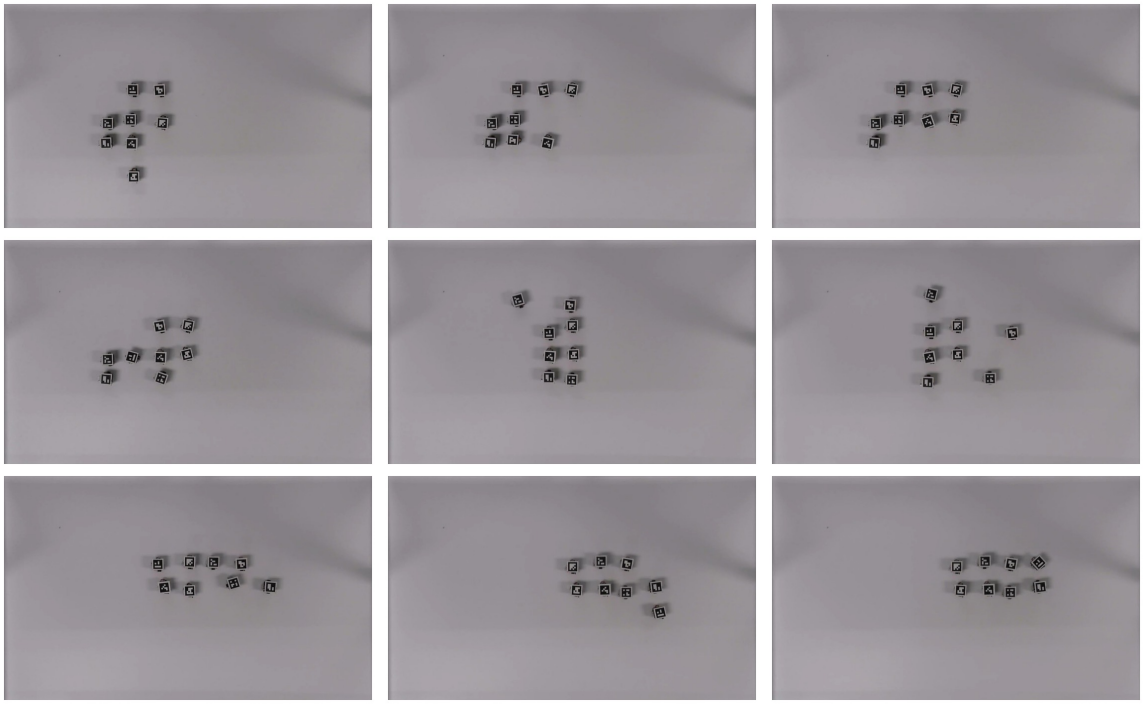


Figure 2: Example of a self-reconfiguration trial on a team of eight GRITSBots. The full image sequence is shown in Chapter 6.

However, all of these applications require scalable and adaptive methods capable of controlling large numbers of modules. Chapter 5 therefore addresses the limited adaptability of the methods presented in Chapter 4 by introducing time-varying learning rates as well as assembly mode switching using different utility functions. Time-varying learning rates allow agents to modify their aggressiveness level with respect to the exploration of the state space (as opposed to greedily exploiting their knowledge for the assembly of the target configuration). An assembly mode switching scheme allow agents to select a beneficial assembly mode based on local state information with the goal of improving convergence rates to the target configuration (compared to the results achieved in Chapter 4). A number of utility functions are presented to exemplify the design of assembly modes and to demonstrate the effectiveness of assembly mode switching. The main theoretical contributions of this chapter are the proofs of convergence for both adaptive methods, which are based on the theory in Chapter 4.

All of the methods introduced so far have been validated extensively in simulation.

Ultimately, however, what makes a self-reconfiguration algorithm useful is its applicability to robotic systems. As such, Chapter 6 leaves the sheltered existence of simulation and instantiates the algorithms developed in Chapter 4 and Chapter 5 on robotic hardware (an example is shown in Fig. 2). Specifically, a novel multi-robotic testbed is presented that allows the execution of self-reconfiguration algorithms. While developed with self-reconfiguration in mind, this testbed - dubbed *the Robotarium* - aims at being a much more general, usable, and accessible research instrument. The Robotarium contains miniature wheeled robots - the *GRITSBots* - which were specifically designed for compact multi-agent experiments [144]. While Chapter 6 focuses on showing the feasibility of our two-dimensional reconfiguration methods on the Robotarium, it is just one of many example application supported by the GRITSBot. Robotarium users have instantiated algorithms for tasks as varied as consensus, cyclic pursuit, leader-follower networks, formation control, and coverage control.

Altogether, the research in this work seeks to advance the field of self-reconfigurable robotics through novel and scalable control methods on the one hand and an exemplary instantiation on a custom-designed yet general-purpose multi-robot testbed on the other hand. The underlying theory based on graph- and game-theoretic tools is suitable for a wide array of problems that rely on decentralized local control to guarantee globally emerging properties.

Chapter II

BACKGROUND

This section introduces the concept of self-reconfigurable systems, their properties, potential, challenges, opportunities, and ultimately the underlying goal of developing and using them in real-world applications. We will survey existing architectures, present common abstractions used in modeling these systems, discuss popular control approaches that are capable of handling such high degree of freedom systems, and outline applications and opportunities. While self-reconfigurable systems have great potential, a number of research questions need to be resolved before they will find widespread use in real-world applications. Therefore, this section also presents an overview of the unsolved challenges these systems face. In concluding this chapter, we also define the self-reconfiguration problem in a rigorous fashion and detail the specific abstraction used to represent self-reconfigurable systems in this work.

The main purpose of this chapter is twofold. On the one hand, it serves as a characterization of self-reconfigurable systems and introduces their appealing properties that could make their use beneficial for a wide range of applications (assuming that the challenges mentioned in Section 2.1.5 can be addressed). As such, this chapter aims to survey the rich literature on self-reconfigurable systems and embed the presented research therein. On the other hand, this chapter provides a general problem setup used throughout this work with chapter-specific modifications.

2.1 Self-reconfigurable Systems

Self-reconfigurable systems (SRS) are composed of (large numbers of) physically interconnected modules. A module can be thought of as the basic building block of SRS much like cells are the basic building block of biological systems. Unlike their biological counterpart however, modules are capable of computation and in most cases actuation as well (though robots with passive modules exist). The great potential of SRS arises from the fact that their morphology or shape is not fixed but can be modified at any time by changing the

connectivity of their modules. Modifying their structural configuration allows SRS to adapt to multiple functional requirements, tasks, or environmental conditions. Unlike traditional robots, which are designed to perform a specific task, SRS are designed bottom-up, where a basic module design enables various geometries, functions, and capabilities of the aggregate system. The potential of self-reconfigurable system arises from a number of different dimensions such as robustness, fault-tolerance, extensibility and modularity, versatility and adaptability, low-cost, and resource reuse (see [101, 196]).

Robustness Another potential advantage of self-reconfigurable systems is their inherent robustness to perform a task despite partial failures. Whereas conventional robots are often rendered nonfunctional in the face of hardware failures, SRS can eject and replace broken or malfunctioning modules (assuming an oversupply of modules in the system). The capability of autonomously replacing faulty parts leads to a notion of self-repair (see [196]).

Versatility and Adaptability One of the biggest advantages of self-reconfigurable systems is their potential for versatility and adaptability. Their ability to dis- and reassemble allows them to form new morphologies that are better suited for new tasks and new environments. For example, a robot using legged locomotion can reconfigure into a wheeled configuration. Regarding manipulation, a robot could, for example, extend or shorten its manipulator arms and end-effectors during runtime, or add/remove arms altogether. As such, SRS support multiple modes of locomotion, manipulation, and also perception. However, SRS are likely to be less efficient when compared to monolithic task-specific robots. For example, a wheeled robot is likely to be able to drive faster than an SRS in a wheeled configuration (see [101, 196]).

Extensibility and Modularity Extensibility and modularity of self-reconfigurable systems was highlighted in [101]. Whereas the traditional design paradigm of robots emphasizes designs with a minimum number of components, SRS are built on the premise of an abundance of modules. This makes SRS inherently extensible systems, since additional modules allow the aggregate system to perform a larger set of tasks or execute them in a more parallel

fashion. The minimal design of traditional robots on the other hand, limits their ability to adapt to new tasks or extend their functionality. As an example, imagine a scenario where an armed manipulator tries to grasp an object but fails because the object has a too irregular shape. A traditional robot would fail for lack of adaptability. A modular robot on the other hand can simply change its gripper geometry or assemble a new manipulator altogether.

Low Cost and Mass Production In [196], Yim et al. argue that economies of scale and mass production of modules could work in favor of self-reconfigurable systems. This could particularly prove true for SRS that are composed of a single type (or few types) of modules, in which case, large numbers of modules could be built at low cost. Additionally, SRS allow resource reuse in the sense that they can be disassembled after a task is completed and their modules reused for a different purpose.

The use of self-reconfigurable systems however does not come without certain disadvantages. For example, the inherently large number of modules increases the mechanical complexity of the overall system as well as the computational complexities of controlling an SRS. However, the main point of criticism of SRS is the fact that such a modular robot is likely inferior at performing a certain task than a robot specifically tailored, designed, and built for that task. In that sense, SRS will be most advantageous and live up to their promise in scenarios, where multiple tasks need to be addressed that would otherwise require a set of fixed-morphology robots (see [196]). The following sections present a taxonomy of SRS, commonly used abstractions and models, a review of controls approaches, as well as current and future applications. A more detailed discussion of the challenges that self-reconfigurable robots face will then be presented in Section 2.1.5.

2.1.1 A Taxonomy

Self-reconfigurable systems can be differentiated along a number of dimensions: size, shape, geometry, homogeneous versus heterogeneous systems, active versus passive actuation, etc. Following the general consensus in the literature, we present a classification by geometry which includes lattice-type, chain-type, and mobile systems, as well as other types that do not fit in either category. This section intends to give an overview of the major categories.

A comprehensive list of historic and current hardware implementations is shown in [71, 131, 175, 196]. Note that the reconfiguration approaches in Chapter 3 to 5 are applicable to lattice-based systems and are then mapped to the dynamics of a mobile system in Chapter 6.

Lattice-type Systems In lattice-based systems, units are arranged and connected in a regular two- or three-dimensional grid pattern. Cubic lattices are most commonly used ([5, 23, 24, 70, 73, 155, 154]), but other types such as hexagonal grids ([190, 44]) or triangular grids ([94]) have also appeared in the literature. In general, module geometries that create regular periodic grids require a certain degree of symmetry. Murata et al. [131] compare the modules in these systems to cells in biological systems or to the atoms in crystal.

Lattice-based systems are appealing from a simulation perspective because they allow abstractions that facilitate generic reconfiguration algorithms. The reason for this is that the lattice type implicitly specifies the allowable motions on that lattice. Furthermore, modules are only able to move to a discrete set of adjacent lattice positions. This discrete nature of possible module motions allows them to be open loop because they do not require precise relative alignment between modules. The actuators and connectors embedded in the modules have to enable them to execute motions and connect to adjacent modules either in a self-actuated fashion or with the help of other modules. This is where a high degree of symmetry is disadvantageous because the higher the symmetry, the higher the number of degrees of freedom and the higher the number of actuators and connectors required to accomplish module motion. This in turn leads to high hardware complexity, complicates the design, and leads to low power-to-weight ratio of modules (see [71, 131, 196]).

For these reasons, hardware implementations tended to be large and bulky ([106]), operated in two-dimensional workspaces (the Crystal system[160] or EM-cubes [5]), or relied on external actuation (the Miche and the Pebbles systems [72, 73, 74], or systems suspended in fluids [192, 136, 187]). Most lattice-based systems featured modules that could only execute motions with the help of other modules (for example the Molecule system [100]). Only recently, a cubic hardware implementation was presented that was capable

of full three-dimensional motion with few actuators ([154, 155]). The 3D M-Blocks rely on a single fly-wheel actuator that allows them to locomote in three dimensions while the connection mechanism uses permanent magnets. It remains to be seen whether this novel hardware leads to increased popularity of lattice-based systems in physical implementations. In the simulation domain, lattice-based systems already enjoy a certain popularity because of the relative ease and compactness with which lattice-based systems can be computationally represented and controlled (compared to chain-type systems operating in a continuous domain).

In the presented work, we exclusively develop methods for lattice-type systems containing modules that are arranged in a regular grid, more specifically in a cubic grid. Details about the particular abstraction used in this work are shown in Section 2.3.

Chain-type Systems Chain-type systems consist of modules that are connected in a serial chain of actuated joints and links between joints. Depending on the number of connectors on each module these topologies can either be purely serial (in the case of two connectors per module), create tree topologies or even include cycles (in the case of branching modules with more than two connector interfaces). Tree-like structures allow the system to create multi-limbed structures with a variable number of variable-length limbs, which makes this type of self-reconfigurable system appealing, for example, for space applications. An advantage of using chain-type systems is that they can fold up to become space filling, i.e. emulate lattice-based systems. So even though the underlying architecture is serial, these systems can arrange modules in a grid pattern. On the other hand, lattice-based systems can appear like chain-type systems if modules are connected in a linear chain (see [71, 131, 196]). Intuitively, chain-type systems can be thought of as n-link kinematic chains or serial manipulators with a variable architecture. Tree configurations appear similar to parallel manipulators though they do not necessarily have a platform or an end-effector as the central component.

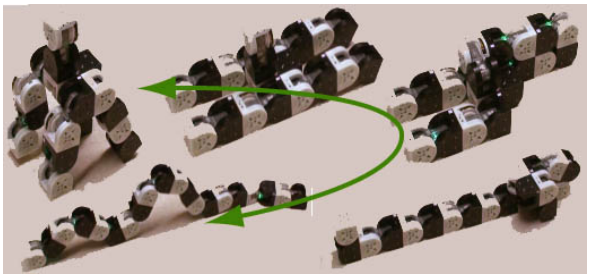
Chain-type systems typically have a lower degree of symmetry, which means that module

designs require fewer actuators and connectors to be functional. The resulting lower hardware complexity goes hand in hand with a higher power-to-weight ratio, which means that motion generation is more easily accomplished than on lattice-based systems. However, the continuous nature of module motions complicates self-reconfiguration of chain-type systems because precise alignment between modules is required to establish a connection between modules. As such, chain-type systems tend to be more difficult to control, more complex to represent, and harder to analyze. Yet they offer the potential to reach any point in the configuration space and, therefore, are potentially more versatile than lattice-based systems (see [131]). Popular configurations of chain-type systems include legged topologies, wheeled configurations, or snakes.

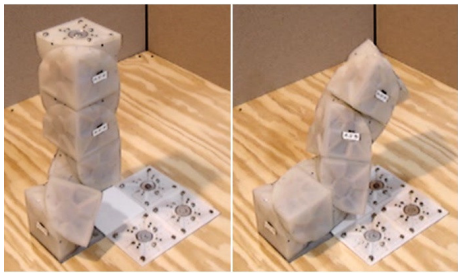
The lower degree of symmetry present in most chain-type systems facilitates hardware design, which is one of the reasons, the literature feature a number of physical realizations of chain-type systems. Examples include the CONRO system (see [39, 38, 169, 168, 177]), the various iterations of the M-TRAN system (M-TRAN I [108, 110], M-TRAN II [107], and M-TRAN III [131, 109], see Fig. 3a[1]), the SuperBot (which can be thought of as a modified M-TRAN module with an added degree of rotational freedom [167, 163]), the CKBot ([141]), the Molecube ([203, 202], see Fig. 3b[2]), or the Roombot ([174, 173]). What all of these systems have in common is their relatively low number of degrees of freedom per module (up to two DoF according [196] with the exception of the SuperBot, which has three DoF). These modules, however, allow the formation of complex structures with significant flexibility. Another chain-type system is the Milli-Motein system [98]. While most self-reconfigurable systems allow modules to connect and disconnect from one another, the Milli-Motein's topology is fixed and self-reconfiguration happens through changing the folding schemata similar to protein folding in biology. Related, though purely in simulation, is the work by Cheung et al. on universally foldable strings [42].

[1]Image courtesy of H. Kurokawa (AIST), licensed under Creative Commons Attribution 2.5 Generic

[2]Image courtesy of Victor Zykov (Cornell Computational Synthesis Lab), licensed under Creative Commons Attribution-ShareAlike 2.5



(a) Shape metamorphosis shown on the M-TRAN III system developed at AIST (Advanced Industrial Science and Technology, Japan)



(b) The Molecube system developed at the Cornell Computational Synthesis Lab.

Figure 3: Examples of hardware instantiations of chain-type systems.

Mobile Systems The modules of this type of self-reconfigurable system do not necessarily depend on other modules to move but can instead use environmental features for locomotion. These modules can form chains of modules as well as complex lattice patterns or alternatively move through the environment completely independently from other modules. An example of a robots that falls into this category is ModRED system [81] or the distributed flight array [138] that consists of units that are capable of wheeled locomotion on the ground and rely on other modules to fly (since each module houses only a single rotor, see Fig. 4[3]). The system presented in Chapter 6 also falls into this category of systems.

Other types Not all systems that have been proposed over the years allow the above categorization. Truss systems, for example, are neither of the above types of systems. They contain active elements or struts that achieve deformation through contraction and expansion. These active elements are connected through passive links and joints. Examples of truss systems include Morpho [199] and Odin [116]. Intuitively these systems form objects that bear similarity to scaffolding. Another type that does not fit the conventional categorization are free-form systems such as the MEMS-based devices shown in [51] and [52]. These MEMS devices are controlled through voltage pulses that allow them to move in the plane. They can also be controlled to dock and form larger planar structures. Catoms, for example, ([77, 93, 76]) represent another system that could be considered free-form. These

[3]Image courtesy of Raymond Oung (ETH Zurich), licensed under Creative Commons Attribution-Share Alike 3.0 Unported

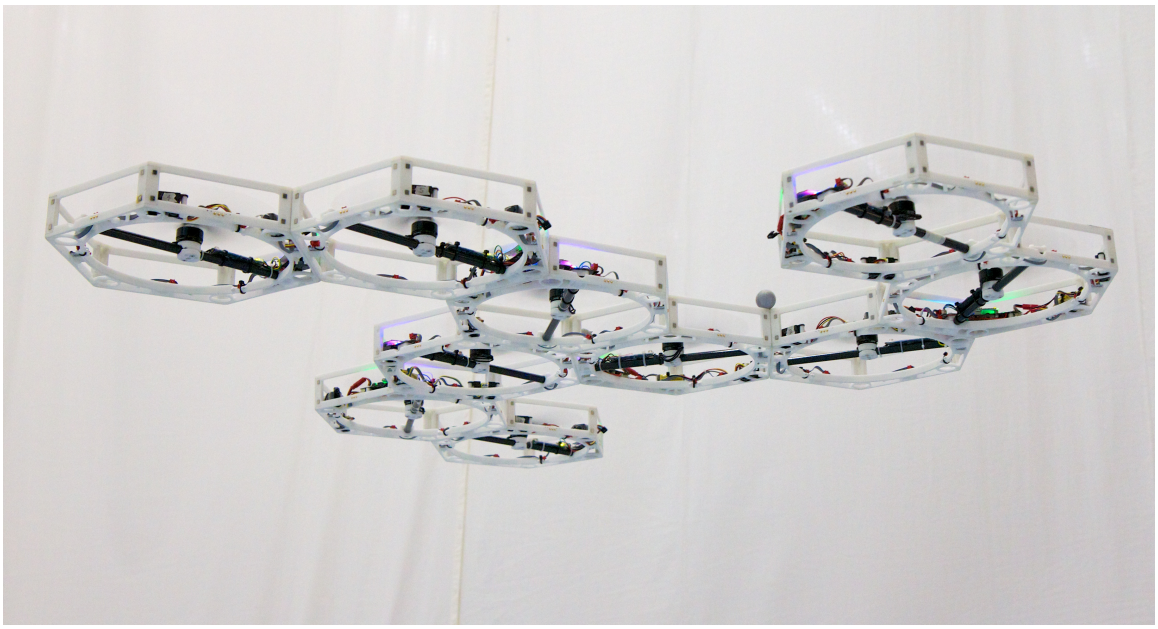


Figure 4: A 10-module configuration of the Distributed Flight Array as an example of a mobile self-reconfigurable system.

cylindrical modules are actuated by 24 electromagnets arranged around their circumference. While capable of forming regular lattices, these modules are also fully functional without a lattice.

2.1.2 Abstractions and Models

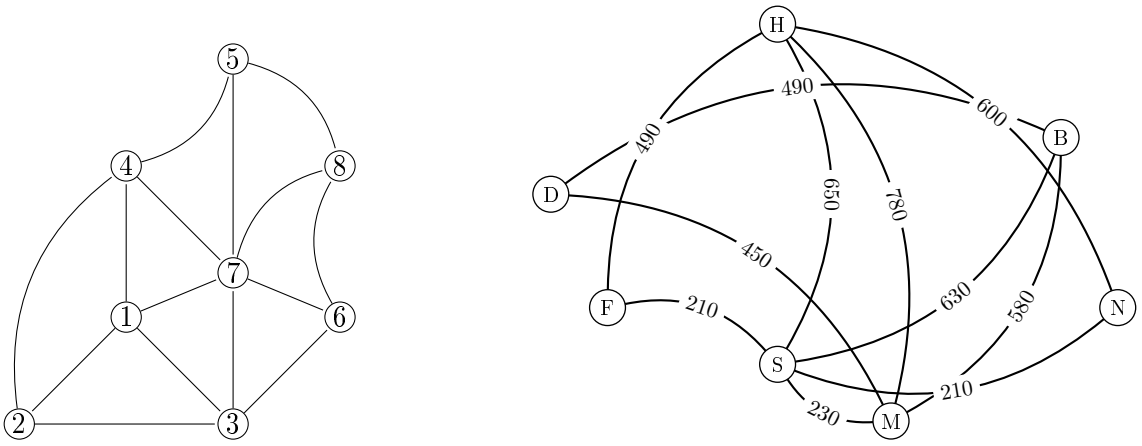
Numerous methods of representing self-reconfigurable systems have been proposed over the years. In this section we want to provide a brief overview of the most commonly used abstraction methods of self-reconfigurable systems. Unlike [1], which provides a thorough and detailed list categorized by solution methods, this section groups abstraction methods by the underlying data structure. As such, we categorize the various abstractions into graph-based representations (connectivity and connector graphs, lattice connectivity graphs), matrix-based representations (incidence matrices), discretization and grid-based methods, and geometric approximations.

In general, abstractions facilitate the control of complex systems. In the case of self-reconfigurable systems, abstractions allow to decouple the complexities of transitioning between states from the self-reconfiguration algorithm itself (for example in the sliding cube

model described below). A module motion planner, for example, does not need to know the details of how a primitive motion is executed, it only needs to know the set of possible motions. The reconfiguration planning methods of Chapter 3 as well as the agent-based methods in Chapter 4 rely on such an abstraction of the motion model. In Chapter 6 we then show how abstract motion primitives can be mapped to the continuous dynamics of robotic hardware.

Graph-based representations Graph-based abstractions represent a self-reconfigurable system as a set of vertices and a set of edges, where a vertex represents a module and an edge represents a connection between two modules. These *connectivity graphs* (or module graphs as they were initially called in [36]) represented a configuration of modules as a graph, which made it accessible to graph-theoretic tools and efficient solution methods. In [140] for example, a graph-theoretic similarity metric between different configurations was proposed that relied on a similar connectivity notion called the *lattice connectivity graph*. This metric could then be used to estimate the number of module motions to transition from one configuration to another and determine whether two configurations are equivalent. One of the main downsides of using connectivity graphs, however, was the fact that different configurations could have the same graph topology (see [1]) because connectivity graphs were unlabeled. This issue was addressed in [82] through labeled graphs (more specifically edge labeled graphs that described the connector state of modules) and directed graphs that encoded additional information in the direction of edges ([38]). Later work by Asadpour et al. ([8, 9]) used the notion of graph isomorphism to determine whether two configurations were equivalent and if not, determine their dissimilarity. They introduced the notion of graph signatures that were based on connectivity information between modules and labeled edges. These labeled graphs could then be used for configuration discovery through graph isomorphism checks, i.e. a self-reconfigurable system could determine how its modules were currently arranged.

In general, graph-based abstractions are compact and concise representations of a system and are well suited for the application of search-based solution methods (see [1]). Because



(a) Example of an unlabeled graph.

(b) Example of a labeled graph containing both edge and vertex labels.

Figure 5: Examples of unlabeled and labeled graphs.

the complexities of a system are abstracted away and encoded in a set of vertices and edges that can be traversed easily, these search-based methods tend to be fast and efficient. The approaches shown in Chapter 3 rely on a graph-based representation of the system but also make use of notions from grid-based abstractions shown below.

Matrix-based representations Matrix-based representations are closely related to graph-based ones in that they also encode the connectivity information of a self-reconfigurable system in a structured form. *Incidence matrices* (see [40]), for example, are the matrix equivalent to connectivity graphs. An incidence matrix is an $n \times m$ matrix where n is the number of vertices (or modules in the system) and m is the number of edges (or the number of connectors/joints of a module). Here, an entry $a_{i,j} \in \{0, 1\}$ indicates the absence or presence of a connection between link i and joint j. The incidence matrix notation's main disadvantage is that it encodes no information about the connectors' docking orientation and the permutation in which connections are established. It therefore does not fully describe a configuration. A slight twist on the notion of incidence matrices was introduced in [41] through *assembly incidence matrices* that replaced 0 and 1 entries by port numbers that described through which port of a connector a module i was connected to another module j.

This modification allowed the identification of isomorphic configurations and the introduction of equivalence classes of configurations. A specialized matrix notation for truss-based systems was introduced through the *configuration coupling model* that used an $n \times 3$ matrix to describe a configuration (see [53]).

Grid-based representations A number of methods rely on grid-based representations of self-reconfigurable systems and the environment they are embedded in. Two methods in this category are the sliding cube model initially introduced in [28] and the pivoting cube model [184]. Both models represent modules as cubes in a discrete regular lattice. These cubes can occupy positions in the grid and move between grid cells using primitive motions. While the sliding cube model allows for two primitive motions - sliding motions and convex transitions around other modules - the pivoting cube model only offers a single motion primitive. Modules are only allowed to pivot (rotate) around an edge shared with another module.

The sliding cube model has been used extensively since its introduction (for example in [30, 61, 64, 145, 146]) and instantiated on a number of hardware implementations - most notably the Superbot system [163, 167], the M-TRAN system [130, 108, 107, 110, 131, 109], and similar systems such as the Crystal [160] and the Telecube robots [189]. The pivoting cube model was introduced only recently but already found application on the M-Block [154] and 3D M-Block system [155].

The main advantage of these models is that they decouple the complexities associated with executing low-level motion primitives from the actual self-reconfiguration algorithm itself (see [1]). Note that in this research, we will exclusively use the sliding cube model and impose a varying number of constraints on the system depending on the specific self-reconfiguration algorithm used. A detailed description of the sliding cube model can therefore be found in Section 2.3. Additionally, in Chapter 6 we will show how a high-level self-reconfiguration algorithm can be instantiated on a physical system using a low-level control layer responsible for executing primitive actions.

Approximation-based representations The main goal of approximation methods is a compact description of a (target) configuration either by using a *volume* or a *surface approximation*. The approach is similar for both methods in that a geometric expression is used to describe a configuration. For the volume approximation method, a configuration can be represented by bounding boxes of equal size (as shown in [63, 4, 70]) or of variable size (as shown in [182, 179]). Clearly, the quality and accuracy of such an approximation depends on the number and sizes of bounding boxes used. Larger numbers of small boxes increase the accuracy of this approach (see [181]). The surface approximation approach on the other hand characterizes a configuration by its external surface (see [1]). This surface can be thought of as partitioning the space into an interior and exterior subspace. A module's goal is to occupy a position in the interior subspace. Therefore, for this approach to work, modules need to be able to determine in which subspace they currently reside. This is a tried and tested approach in computer graphics, where volumes are often approximated using triangular surfaces. However, it is not clear that modular robots with limited processing power are able to handle this computationally intensive method (see [181]).

Other methods A variety of other types of representations exist in the literature. Here, two loosely biologically inspired examples are presented that contrast the approaches introduced above. Both of the following approaches are based on folding. Whereas Cheung et al. [42] introduce a method that folds one-dimensional strings into two- and three-dimensional structures similar to protein folding, Nagpal et al. [135, 134] fold two-dimensional sheets of modules into two- and three-dimensional objects. The latter method was inspired by paper folding (origami). Hardware implementations already exist for both types. Foldable strings of modules have been presented in [98] (called the Milli-Motein) whereas foldable robots have been presented in [127]

2.1.3 Control Approaches

A recent overview paper [1] identified nine basic operations for modular robots, which included self-reconfiguration, self-assembly, self-disassembly, self-adaptation, grasping, collective actuation, flow, gait, and enveloping. In this section, the focus lies on self-reconfiguration

with tangential coverage of the neighboring fields of self-(dis)assembly and flow. Specifically, in this work, we treat flow (or locomotion through self-reconfiguration) as part of self-reconfiguration. More generally, the approaches presented in this document allow for self-assembly, self-disassembly, self-reconfiguration, as well as flow. This sections loosely follows the categorization shown in [1] with a focus on search-based and agent-based methods, as the methods shown in this work can be categorized as such.

Note that control approaches could also be categorized according to a number of other attributes such as deterministic versus stochastic control (see [196]), centralized versus decentralized control, serial versus parallel module motion, or online versus offline planning (precomputation). In this section however, we focus on a categorization based on the underlying solution mechanism.

Search-based methods Search-based methods have their origin in the artificial intelligence domain where for example Russell et al. [161] define a search problem as having five basic components: an initial state, a (state-dependent) set of actions, a transition model, a goal test, as well as a path cost. The first three elements implicitly define a state space, which in turn gives rise to a directed transition graph in which nodes are states and edges are actions that lead from one state to another. The goal of a search-based method is to find a sequence of actions from the initial state to a desired goal state (i.e. the target configuration) through the state space. Applied to the self-reconfiguration domain, a state in the state space is that of an entire configuration, a path through the state space is a sequence of configurations, and actions are taken by individual modules that change the configuration.

The most elementary search methods do not require any knowledge about the search problem beyond the above definition of the five key elements. This group of algorithms include depth-first search, breadth-first search, or Dijkstra's algorithm (see [161]). The lack of additional knowledge about the search domain, however, introduces a high branching factor and a state space that is exponential in the number of modules, which makes uninformed search intractable for general self-reconfiguration problems. According to [1], one of the tractable scenarios occurs when the size of the configuration graph (i.e. the state space) is

small enough (see [191]). Alternatively, search methods can be used as an auxiliary component, for example, for computing a module's path from one lattice position to another (as done in [100, 160, 63, 112]).

Search methods that have access to specialized knowledge about the search domain allow a guided search towards promising directions or states that are likely to lead to goal states. These methods are called informed search methods and include algorithms such as A* or D*, which are both optimal and complete. In other words, they are guaranteed to find the shortest path from an initial state to the goal state. A*, for example, has been applied to the self-reconfiguration problem in [13, 145, 146], D* in [19]. These search methods rely on a heuristic function to estimate the distance of a state to the goal state. Various such heuristics have been proposed in the literature (for example in [140, 9, 8]). While originally developed as centralized methods, all of the above mentioned informed and uninformed methods can also be implemented as distributed algorithms but then rely extensively on information exchange via message passing. Methods that were specifically developed for distributed execution are shown in [46, 47]. These methods equip each module with the capabilities to plan its own actions based on local information gathered via local perceptions or local communication.

While the above methods work in a deterministic fashion, random and stochastic approaches have also been applied to the self-reconfiguration problem, for example, Probabilistic Roadmaps (PRM, [87]) or Rapidly-exploring Random Trees (RRT, [113]). RRTs generate random trees rooted in the initial configuration and have been shown to find solutions faster than A* given that the self-reconfiguration problem was sufficiently difficult (see [25]). Improvements to RRT-based methods with respect to the Superbot and M-TRAN modules were studied in [78]. Another probabilistic method is based on simulated annealing and has been employed in [140, 43].

In this work, search-based methods are employed in the centralized approaches of Chapter 3 to compute individual module paths to goal locations as well as in Chapter 4 (Section 4.2) to verify the groundedness of modules.

Agent-based methods An agent, according to [161], is "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." This fairly broad definition applies to software-based agents, but can also be applied to physical and simulated robots, for example, modules in a self-reconfigurable system. In this context, a module interpreted as an agent can be viewed as an autonomous rational entity that can effect change in the environment through actuators and perceive the world through its onboard sensors. Agent-based methods give rise to a natural way of distributing the intelligence and decision making authority in a self-reconfigurable system since every agent has to make decisions based on local information that it can gather via its own sensors or communication with other autonomous agents. From an agent's perspective, the world can be partitioned into two components: the agent's interior state and the environment (or everything that is external). In that sense, the environment includes other agents that impose kinematic constraints upon each other. This category includes a cornucopia of solution approaches (see [1]), for example, local rule-based methods ([86, 26, 146], cellular automata [29, 31, 28, 30, 178], MDP-based formulations and dynamic programming [60, 64, 162], reinforcement learning-based methods [171], game-theoretic methods [153, 152, 54], and leader-follower approaches applied to self-reconfiguration [115, 83].

Here, we will focus on rule-based and game-theoretic approaches since the methods in this work and specifically in Chapter 4 are based on these techniques. Rule-based methods including cellular automata induce finite state machines, in which an individual agent determines its next action based on its current state and the state of its neighbors. Cellular automata have been introduced by John von Neumann in the 1940s and were first applied to self-reconfiguration in [29]. In that particular work, manually designed rules enabled a self-reconfigurable system to locomote through environments with obstacles. Further extensions in [28, 30, 31] allowed the system to split and merge as well as self-replicate. While the rules used in these publications were manually designed for a particular tasks, a number of approaches for automatic rule-generation exist. Rule generation for two-dimensional reconfiguration has been demonstrated in [86], rules for the three-dimensional case were covered in [146, 178] (and also Chapter 4), and auto-generated rules were applied to the ATRON

system in [26]. While local rules are computationally cheap to verify and apply by an agent using only local information, their main disadvantage is that they have to be precomputed or manually designed before an agent can use them. As such, their adaptability to changing environmental conditions is limited.

Game theory is widely used in the field of multi-agent and swarm robotic system with a rich body of work to tap into. More generally, game-theoretic methods have been applied to numerous academic and non-academic domains including modeling of stock markets, resource allocation in networking, behavioral psychology and modeling of biological systems, artificial intelligence and decision making, or politics. Specifically in the multi-agent domain, game theory found applications in vehicle-target assignment [7], coverage control [114, 119], sensor-deployment problems [119], the consensus problem [119], congestion games and traffic modeling [120], and numerous others. However, game-theoretic formulations of the self-reconfiguration problem remain few. To date, self-reconfiguration has only been formulated as coalition games, which incentivizes agents to form and remain in teams (see [153]. This coalition game structure was then used in [152] as a connectivity preserving mechanism which allowed further coordination towards assembling the target configuration. In [54, 55], Dutta et al. exploit coalition forming to find the best configuration of a modular robotic system to perform a given task efficiently. However, other game-theoretic tools such as potential games or learning algorithms such as log-linear learning have not yet been applied to the self-reconfiguration problem.

In Chapter 4, we will show how a novel game-theoretic formulation can solve the self-reconfiguration problem in a decentralized fashion while guaranteeing global properties. Modules will be represented as autonomous decision makers that select their actions based on local information. Yet, we will be able to guarantee the assembly of the target configuration, which is a global property of the aggregate system.

Biologically inspired methods Bio-inspired methods are similar to agent-based methods in that they assume a module to be a self-contained entity capable of autonomous decision making such that the aggregate system exhibits some desirable global behavior.

The field of biologically inspired multi-agent control algorithms for self-reconfiguration is too varied to cover in sufficient detail and breadth. This section therefore only aims at providing exemplary evidence while a more thorough overview is provided in [1].

In [79], for example, a hormone-based control approach is presented that uses evolutionary computation to adapt controllers to environmental conditions. A related approach borrows ideas from biology through a hormone-inspired communication protocol in [168]. Also related to hormone-inspired control are gradient-based methods where a control policy guides modules to follow a gradient to a goal position (see [1]). The complexity in these approaches is shifted to computing gradients as opposed to computing local control signals. Seed nodes are used in [178] that act as attractors similarly to artificial chemical. Modules then compute a concentration of these artificial chemicals and follow a gradient in a steepest descent fashion. Similar methods have also been applied in [176, 180].

One last class of algorithms that falls both into the category of probabilistic search methods as well as bio-inspired methods are genetic algorithms, which maintain populations of solutions and then iteratively propagate the fittest solution through reproduction and selection. For example, genetic algorithms have been used to evolve emergent behaviors [111] and controllers [183], but also for adaptive action selection [90]. In connection with simulated annealing genetic algorithms have been used to solve multi-objective optimization problems [53].

Nano-inspired methods The idea of micro- and nano-assembly was proposed as far back as the 1960s by Richard Feynman [58] who hypothesized about the possibility of manipulating matter on an atomic level. A thorough review of applications ranging from manufacturing techniques to micro robots is given in Chapter 18 of [172]. An excellent overview paper [20] specifically focuses on self-assembly at the meso-scale (microns to millimeters). Methods that cover the smaller range of nanometers to microns are presented in [193] together with a general outline of the effects influencing self-assembly at such a scale. The remainder of this section provides a brief introduction of nano-inspired methods and focuses on a select few publications relevant to the field of self-assembly on a micro- and nano-scale level. Note

that all presented systems in this sections contain passive building blocks or particles. However, the advent of MEMS (microelectromechanical systems) is beginning to change passive towards active modules (see Chapter 18 of [172]).

Common themes in nano-inspired methods include the use of anisotropy of particles, interaction of heterogeneous particles, the lock-and-key principle of interlocking parts, and the design of interaction forces that in turn determine which configurations can be assembled. In [126], for example, local rules are designed based on heterogeneous shapes of parts and varying attractive forces (in this work this property is referred to as patchiness) that give rise to planar assemblies of parts. These local rules are similar to the cellular automata-based and rule-based approaches which were mentioned above for agent-based control strategies. An overview of the parameters that can be used in the design of rules was shown in [188], which include curvature radius, patch size, aspect ratio, roughness, and a host of others.

Similar work on the tunable self-assembly of colloidal crystals using reconfigurable building blocks was investigated in [99]. Design parameters in this paper are the size ratio of overlapping spheres that building blocks are composed of as well as tunable interaction forces between particles, for example, depletion or attraction forces.[4] A related paper elaborated on the lock-and-key principle used for assembling colloidal molecules and included a richer model that allowed for more fine-grained control over the reconfiguration of building blocks. While these approaches do not allow for runtime modifications, more recently a method enabling the real-time controlled assembly of colloidal particles (insoluble particles suspended in another substance) was shown in [12]. In this work, a high-dimensional particle system is observed through real-time imaging sensors. This sensor data then informs the design of a Markov Decision process-based control policy that controls the assembly process through externally applied force manipulation. This approach presents an externally actuated, centralized self-assembly process with passive parts. In this work however, we focus on active agents capable of autonomous decision making and self-actuation based on local information.

[4]Note that tunable parameters are meant in a sense that rules can be manually tuned during the design phase. No real-time adjustments during the assembly process are possible.

2.1.4 Applications and Opportunities

Even though the first self-reconfigurable system was proposed in the late 1980s ([67]) and algorithmic and theoretic results are already mature to the point where millions of modules can be modeled and controlled, hardware implementations are still lagging behind. The lack of general purpose hardware modules (as mentioned in Section 2.1.5) complicates application development and currently requires specialized hardware for every application. Nonetheless self-reconfigurable systems show great promise for the future. As we will see in this section, there already exist numerous prototypes for diverse applications. Up to this point however, no "killer" application has emerged (see [196]).

Exploration One can imagine any number of futuristic application of self-reconfigurable systems. In this section however, we will focus on those applications for which prototypes already exist or will materialize in the not too distant future. In general, suitable applications will exploit, in one way or another, the key features of self-reconfigurable systems - adaptability, robustness, and low cost (once a general module can be mass-produced). Adaptability and robustness carry significant importance in environments which are not easily accessible or offer limited opportunity for human intervention. A lack of accessibility can stem both from a physical restriction (such as small scales in the micro and nano domain, great distances in space, or remote, hostile, and inaccessible environments) or artificial restrictions such as cost. It is, for example, prohibitively expensive to deliver fixed-geometry robots that have been designed for specific tasks into space or to explore deep sea regions. Additionally these environments require adaptability to unforeseen circumstances (see [196]). Space exploration as an application well-suited for modular robots has been mentioned in ([131, 195, 196]). It requires autonomous robots that are self-sustaining over long periods of time and can adapt their functionality without human intervention, self-repair in case of broken modules, or simply adapt to unforeseen circumstances. Self-reconfigurable systems are better able to handle tasks that are not known a priori than monolithic robots with fixed configurations because traditional monolithic robots typically require to be modified by human operators if a new task arises. An example of a hardware implementation that is

geared towards exploration through various modes of locomotion is the chain-type Superbot system shown in [165, 166, 167].

Shape Duplication Another group of applications fits the theme of shape duplication and assembly of shapes from programmable parts. Yim et al. [196] call it *bucket of stuff*, Pillai et al. [149, 150] named it the *3D fax*, Goldstein et al. [76] refer to it as *programmable matter* or *synthetic reality*, Yu et al. [200] use the term *dynamic rendering*, and Gilpin et al. [75] simply call it *3D shape duplication*. The underlying idea is similar in all of these approaches: A set of modules is given a user-specified task and assembles into an aggregate system to complete the task. Such a task could simply be the assembly of a desired shape or furthermore using the assembled structure to fulfill a higher-level function. An example of such a higher-level function is the real-time and remote replication of objects with the goal of eliminating the need for virtual and augmented reality gear by allowing the physical realization of computer-generated objects in real time ([76]). Unlike augmented reality methods, programmable matter would allow the physical interaction (even remotely) with computer-generated objects. The 3D fax is a non-real-time method capable of achieving shape duplication with modules depending on external actuation (see [149, 150]). The main obstacle between these ideas and a large-scale hardware prototype is the necessary miniaturization of modules since current instantiations like the robot pebbles [72] are still centimeter-sized. After all, in this scenario, robotic modules are meant to act similarly to cells in biological systems which requires modules at the millimeter or even micrometer-scale (advances at that scale are shown in Section 2.1.3 - Nano-inspired methods as well as in Section 2.1.5). For example, robotic modules would equip a robot with the ability to heal, self-repair ([24, 47, 46, 103, 132, 159, 170, 182]), and re-grow severed limbs ([176]).

Collective Tasks Tasks that can be accomplished with a given robot are generally governed on a hardware level by its size, the specifications of its actuators, and the physical limits of its links and hinges. Specifically for self-reconfigurable robots, there is a tradeoff between the size and power (i.e. forces that can be exerted) of its modules. Smaller modules can be deployed in larger quantities which typically increases adaptability and the number of

configurations that can be assembled. However, decreasing module size comes at the cost of lower power and dictates module designs with lower complexity. On the other hand, larger modules are typically equipped with stronger actuators and hinges but tend to be overly specialized for specific tasks (see [32]). In this section, we therefore review approaches that make use of the collective nature of self-reconfigurable systems in order to increase (or even multiply) the forces individual modules can exert and the tasks the aggregate system can accomplish.

Collective actuation, for example, is proposed in [32]. The authors present a method to assemble large-scale joints and muscle-like actuators with the goal of making actuator capacity and range a function of the system configuration/geometry as opposed to immutable properties of the module design. Collaborative manipulation is shown in [18], where Roombot modules are used for moving passive objects in 3D space. A similar approach is presented in [24] using ATRON modules. Here, modules are assembled into conveyor surfaces and robotic arms for handling objects. A more swarm-robotic twist on collective manipulation limited to two dimensions (collective transport) is shown in [158]. This paper also introduces the Kilobot, a novel miniature robot using vibration as locomotion modality. An approach and a prototype for collective construction of three-dimensional structures has been introduced in [143]. The system labeled TERMES uses mobile robots to assemble structures out of specialized passive building blocks.

Just like shape duplication applications, collective tasks will benefit from smaller and/or more capable modules. At smaller-scales, self-reconfigurable systems will enable applications such as real-time in vivo diagnostics, collective repair of biological systems, or imitation of biological components such as muscles. At a larger scale, self-reconfiguration can aid or replace the traditional approach of erecting and renovating buildings, improve the transport of objects, for example, on manufacturing floors, and add adaptability and versatility to manufacturing in general.

Adaptive Structures This last group of applications focuses on the adaptive capabilities of self-reconfigurable systems in the domains of furniture, transportation, and structures

and buildings. Adaptive furniture, for example, has been explored in [173, 175] together with intuitive user interfaces [17, 139]. An adaptive aerial transportation system called the *distributed flight array* has been presented in [138] which is capable of adapting its lifting power according to changing payload needs by simply adding more rotor modules. In the future, earthquake resistant structures as well as terrain-adaptive bridges and buildings will be enabled by an approach similar to the one shown in [200, 199]. In this work, adaptive platforms are constructed using CONRO modules and controlled using a tensegrity model of cellular structures.

2.1.5 Challenges

Despite the promising outlook and the many applications that could arise from the use of modular self-reconfigurable systems, many challenges remain. Currently, there still exists a disconnect between the size of systems that can be simulated (millions of modules, see [60, 10]) and hardware instantiations of systems (which hovered around 50 modules for over a decade according to [196] and has only recently been extended to 1024 Kilobot modules in [157, 158]). A number of key steps need to be realized before self-reconfigurable systems will be able to fulfill their promises of versatility, robustness, adaptability, and low cost. In this section, we will explore challenges on the hardware front but also regarding simulation, controls, and algorithm-related issues.

At a more abstract level, there remain gaps in the basic understanding of the possibilities of self-reconfigurable systems. While methods, prototypes, and approaches exist on the hardware, controls, algorithm, and software side, one key issue remains to be fully understood: Is there a general shape for a robotic module similarly to cells in biological systems? And if so, which shape should that be? Currently, hardware instantiations are built with a certain set of tasks in mind and no basic, general-purpose building block or module exists. As such self-reconfigurable systems are capable of executing tasks in their specific niche, but have not been fully able to live up to their promise of versatility. The geometry and capabilities of a module however are the most fundamental properties of a self-reconfigurable system. They dictate which aggregate configurations can be built, which motions can be

executed by individual modules and by the aggregate system, and which tasks an assembled system can accomplish. Versatility and adaptability require the use of highly symmetric geometries such as cubes or spheres. High degrees of symmetry however come at the cost of a large number of degrees of freedom and a high control and modeling complexity. This trade-off between geometric simplicity and symmetry often prompts researchers to sacrifice symmetry (or isotropy) to simplify the design of modules (like the M-TRAN module [131, 109]). However, it cannot currently be quantified whether such design simplifications compromise the functionality of an aggregate system comprised of these modules (compared to modules with higher symmetry). What is lacking is a systematic method of determining which tasks can be accomplished given a certain module geometry and actuation strategy (see [131]). Solving this design space challenge will be necessary for the development of a general-purpose module.

Hardware Challenges Hardware challenges of self-reconfigurable systems are mostly related to cost and reliability. The mechanical, electrical, and connector reliability of modules is a key issue standing in the way of widespread deployment of self-reconfigurable systems. Low reliability is the reason for both an upper limit to the number of modules in a system as well as a lower limit on the size of each module. Therefore, current systems do not yet scale up in numbers and down in size to the point where they become useful ([131]). While electrical reliability can be handled well by current mass-production techniques, mechanical reliability cannot be predicted until a sufficient number of modules has been built or until a smaller number of modules has been operative sufficiently long. Connector reliability is addressed by two main approaches: magnetic connectors (using electromagnets or electropermanent magnets) or connectors using a hooking mechanism. Both approaches have been studied extensively for the purpose of self-reconfiguration. Magnetic connectors have been used for the EM-cube [5], Miche [70, 73], robot pebbles [72], M-Blocks [154], and 3D M-Blocks [155]. Hooking mechanisms are used by the ATRON system [84, 24], by M-TRAN [109], [109], by Superbot [163], or the Roombot [174]. No consensus exists as to which method is more reliable. Magnetic connectors offer a mechanism without moving

parts, while hooking connectors establish a stronger connection without requiring energy to maintain the connection. Murata et al. [131] argue that the hooking mechanism used on the M-TRAN III system is faster, stronger, and more reliable than the previously used magnetic connectors, which [196] confirms by labeling M-TRAN III the most robust self-reconfigurable system.

Another key hardware issue is the limiting nature of module geometry since the capabilities, power, and possible tasks of the aggregate system are determined by the geometry and specification of the basic module. Limits on module power, for example, become especially apparent when we examine chain-type systems. The longer a chain of modules, the more force/torque is required by the module at the base. In current systems, the overall functionality is therefore limited by the maximum power of the base module as it limits the maximum length of any chain of modules. An approach similar to one shown in [32] would be required to multiply the forces that individual modules can exert. This system enabled the creation of large-scale joints and the robotic equivalent of muscles. As a result, the maximum actuator force and the actuator range were not fixed by the individual module design anymore but became a function of the topology of the aggregate system. While Campbell et al. do not address the reliability issue, their approach would allow the miniaturization of modules, since the overall forces that can be exerted by the aggregate system is less dependent on module properties but increasingly dependent on ensemble topology.

One last hardware challenge concerns miniaturization of modules to enable a larger number of applications. Miniaturization will require advances in computation, actuation, and power aspects. While the increase in computing power follows Moore's law [129] and approaches exist for miniaturizing actuator sizes (for example through MEMS devices, see [21]), supplying sufficient amounts of power to modules will remain a significant challenge as modules are built at smaller and smaller scales. The main problem here is the limited power density of current technologies, which in turn presents a hard lower bound on the minimum battery size for a required battery capacity. One possible solution is a power grid-like structure that routes power to all modules in the aggregate system through wired connections. Alternatively, SRS will have to face the tradeoff between energy efficiency of

module motion, power density of batteries, and the size of batteries that modules carry onboard (see [101]).

Control and Algorithmic Challenges This class of challenges can be mostly stated as coordination issues of large numbers of modules. A self-reconfigurable system essentially is a networked system whose network structure is dynamic and not necessarily known by all modules. As such, control algorithms for large-scale systems will have to be inherently distributed and able to achieve key functions despite a lack of full state information. Murata et al. [131] identified these key functions as synchronization for motion generation, distributed high-level decision making, as well as differentiation of module roles. While these tasks mostly rely on reliable, efficient, and scalable communication, another set of challenges needs to be addressed that concern parallelism, optimality, and robustness. Yim et al. [196] identified a number of algorithmic challenges including parallel motion of modules in large-scale systems (for both locomotion and manipulation), optimal reconfiguration planning (with respect to time and energy), robust handling of a variety of failure modes (for example misaligned, erratically behaving, or broken units), and a method for determining the optimal configuration for a given task and environment.

A practical challenge that will require a generalized re-definition of self-reconfigurable systems is mentioned in [131] concerning the connectivity of a system. Most approaches currently require connectivity to be maintained at all times, which rules out systems composed of multiple components or systems that merge into one larger aggregate system. However, multiple connected components of modules blur the boundary of what a self-reconfigurable system actually is. Is a connected component its own system or just part of a larger one? The game-theoretic approach shown in Section 4.2 addresses this problem by defining a module as an autonomous agent that is loosely coupled to other agents in the system but allows them to disconnect and merge at any time.

Another challenge is the co-evolution of form and function of self-reconfigurable systems. Two components contribute to this challenge. On the one hand, a general method is required to determine which morphologies can be achieved given a module geometry, which we have

eluded to before. On the other hand, a method is needed to estimate which functions and motions an aggregate system can achieve given a certain morphology. This co-evolution of form and function will require significant computing power to overcome the inherent exponential complexity as the number of modules grows and will remain a significant challenge (see [131]).

Grand Challenges In addition to the challenges outlined above, Yim et al. [196] list a number of grand challenges. These include robustly operating large-scale systems (on the order of thousands of robots), self-repairing systems (such that systems can autonomously recover from damage), self-sustaining systems (such that systems can operate autonomously over long stretches of time), and self-replicating systems (using raw materials that the system can mine itself). With regard to creating programmable matter, Gilpin et al. [71] mention the necessity to miniaturize modules to micro- or even nanometer-sized modules while addressing the inherently stochastic nature of nano-scale systems (see [196]).

2.2 The Self-Reconfiguration Problem

Self-reconfiguration in this work is understood to solve the following problem (see [147]). Given an initial geometric arrangement of modules $\mathcal{C}_I$[5] and a desired target configuration $\mathcal{C}_T$, the solution to the self-reconfiguration problem is a sequence of primitive module motions that reconfigures the initial into the target configuration (an example of such a sequence is shown in Fig. 1). For planning-based solution methods, this means computing a feasible plan of module motions from $\mathcal{C}_I$ to $\mathcal{C}_T$, where feasibility requires all the constraints to be adhered to (at a minimum the motion model and collision constraints but sometimes also connectivity constraints). For agent-based methods (such as the one shown in Section 4.2), which do not use a planning algorithm, finding such a motion sequence requires agents to collaborate through proper incentivization and exploration of the underlying Markov process. Note that most frequently reconfiguration approaches assume fixed initial and target configurations. Section 4.2, however, will show an approach that is able to cope with

[5] A geometric arrangement of cubes is furthermore referred to as a *configuration*.

changing target configurations during runtime. Depending on the level of information about the global state that is given to each module during planning/decision making, we speak of centralized solutions (where modules know the global state of the system) and distributed, reactive solutions (where modules only know their own state and the states of a small set of neighbors).

In this work we assume that all modules share a common coordinate system or frame of reference. A modules state is then given by its position in the shared lattice, but the state can also include any number of additional attributes required by the specific reconfiguration algorithm (such as module type, labels, current operating mode/role, etc.). For example, Fitch [61] adds the space requirements of any intermediate configuration between $\mathcal{C}_I$ and $\mathcal{C}_T$ to the specification of the self-reconfiguration problem. Given this general definition of a module's state, we will point out specific modifications in each chapter.

2.3 System Representation

The Sliding Cube Model In this work, the building blocks of modular robotic systems are visually represented by cubes. As mentioned in Section 2.1.2, this model is commonly referred to as sliding cube model (see [28, 30, 61, 145, 146]), where cubic modules are embedded in a discrete two- or three-dimensional lattice. According to the taxonomy in [196], these systems are categorized as lattice-type systems. The sliding cube model significantly simplifies modeling of self-reconfigurable systems because it decouples modules' transition dynamics and kinematic constraints from the self-reconfiguration algorithms themselves. The motion of cubic modules is simply described by discrete steps from one grid cell in the lattice or environment $\mathcal{E} = \mathbb{Z}^d$ to another.[6] Without loss of generality, the cubes are assumed to have unit dimension. Furthermore, a cube's current state is its position in the lattice $c_i \in \mathbb{Z}^d$. For the planning algorithms outlined in Chapter 3 these cubic modules also feature globally unique identifiers. Furthermore, a collection of cubes is called a *configuration*. Therefore, a configuration $\mathcal{C}$ composed of N cubes is a subset of the representable space $\mathbb{Z}^{dN}$ as shown in [146].

[6] Since we present two- and three-dimensional self-reconfiguration, the dimensionality d will be $d \in \{2, 3\}$.

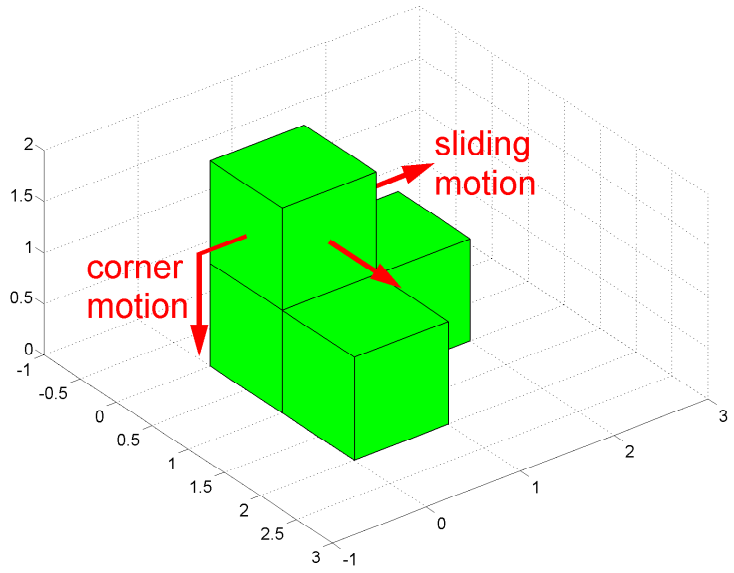


Figure 6: Visualization of motion primitives.

Motion Model In the sliding cube model, a cube features connectors on each of its surfaces[7] and is therefore able to connect to neighboring modules as well as to perform two primitive motions - sliding and corner motions (also referred to as convex or concave motions according to [28]). In general, a motion specifies a translation along coordinate axes and is represented by an element $m \in \mathbb{Z}^d$. A sliding motion m_s is characterized by $\|m_s\|_{L_1} = 1$, i.e. $m_{s,i} = 1$ for one and only one coordinate $i\{\in 1, \dots d\}$, which translates a cube along one coordinate axis. A corner motion m_c on the other hand is defined by $\|m_c\|_{L_1} = 2$ such that $m_{c,i} = 1$ for exactly two coordinates $i\{\in 1, \dots d\}$, which translates a cube along two dimensions (see Figure 6).

Constraints The sliding cube model can incorporate any number of constraints on module motion. In this work, however, we do not impose any physical constraints such as

[7] Therefore, two cubes are considered adjacent if they are connected through a shared face.

module masses, gravity, or other forces.[8] Only collision avoidance between cubes is enforced. Additionally, cubes are required to maintain global connectivity at all times for the centralized methods in Chapter 3 and the graph grammar-based method in Section 4.1. The game-theoretic approach shown in Section 4.2 relaxes the connectivity constraint and allows modules to split and merge at will. Additionally all the approaches shown in Chapter 3, 4, and 5 are able to incorporate space constraints such as walls, floors, ground planes or surfaces, or general obstacles. In Section 4.2 we specifically enforce a *groundedness* constraint, which requires agents to remain connected to a ground plane. Obstacle constraints, or space constraints in general, confine the planning space of the centralized reconfiguration algorithms in Chapter 3 and the action space of individual agents in Chapter 4 and 5. The general problem setup (with case-specific modifications depending on the specifics in each chapter) is the following.

- The environment $\mathcal{E}$ is a finite two- or three-dimensional discrete grid $\mathcal{E} \subseteq \mathbb{Z}^d$ with $d \in \{2, 3\}$.
- N cubes (or modules) move in discrete steps through that grid.
- Without loss of generality, cubes have unit dimensions.
- Each module is capable of sliding and corner motions subject to collision avoidance.

2.4 Conclusions

This chapter has introduced the notion of self-reconfigurable systems and provided an overview of their properties that make them appealing for a range of applications. While these systems show great potential, their widespread use will depend on solving a number of open research questions. These challenges (see Section 2.1.5) span a variety of hardware, algorithmic, and theoretical issues. In this research, specifically, we will address the topics of scalability and decentralization from both a theoretical and an algorithmic point of view. In doing so, we rely on the general definition of the self-reconfiguration problem (Section

[8] These assumptions are made to focus the contribution on the self-reconfiguration process rather than on implementation-specific details.

2.2) and a common abstraction of self-reconfigurable systems called the sliding cube model (Section 2.3). Additionally, we will concern ourselves with the question of how the developed methods can be instantiated on actual robotic hardware. Before doing so, the next chapter will establish key concepts in the domain of centralized self-reconfiguration that serve as the theoretical foundation for the decentralized approaches in later chapters.

Chapter III

CENTRALIZED SELF-RECONFIGURATION PLANNING

This chapter presents centralized methods for reconfiguration planning for homogeneous systems in Section 3.1 and heterogeneous systems in Section 3.2. Self-reconfiguration, as mentioned in previously in Section 2.2, requires to move every cube from its position in an initial configuration $\mathcal{C}_I$ to a position in the target configuration $\mathcal{C}_T$. As such, motion sequences need to be computed for every cube that obey motion and collision constraints. The methods in this chapter rely on a centralized node to compute these sequences in a deterministic fashion. Centralized methods like the planning algorithms presented in Section 3.1.2 and Section 3.2.2 enjoy the benefit of access to global state information of the entire configuration, which facilitates the derivation of global convergence and completeness guarantees. Global information also allows the planner to avoid undesirable states during the reconfiguration process such as deadlocks, holes, and enclosures (as we will see in Section 3.1.3 and Section 3.2.3). From an analysis perspective, global information offers another benefit - it facilitates the characterization of the system behavior on a global level (as opposed to on a module level).

Centralized methods for self-reconfiguration planning have been studied extensively in the literature. The majority of these methods apply to either lattice-based systems or chain-type systems (as defined in Section 2.1.1), which are the system types predominantly used in the literature. The discrete nature of lattice-based systems often favors search-based methods that can take advantage of the finite action sets of agents during planning. Rus et al. [160], for example, present a homogeneous reconfiguration planning algorithm designed for their Crystalline system. A planner tailored to another hardware system, the Molecule robot, was developed in [102, 105], which demonstrated how scaffold planning allowed modules to tunnel through structures. Another discretization-based method utilized the D* algorithm to generate motion sequences for the Roombot modules [19]. Unlike these algorithms that

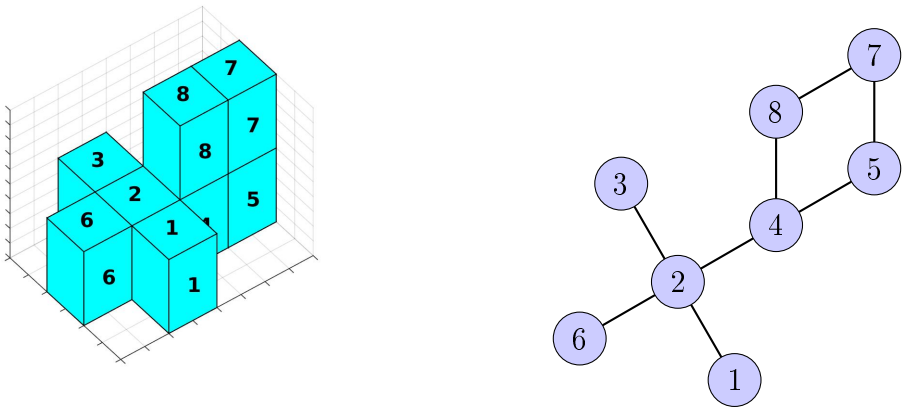
were developed with certain hardware platforms in mind, other work applies to more abstract system models. An algorithm that reconfigures generic two-dimensional hexagonal modules was presented in [112]. Though completeness is sacrificed, this planner manages to solve certain problem instances in $O(N)$ time. Another abstraction, the sliding cube model, is used in [3]. Featuring a time complexity of $O(N^2)$, the planning algorithm in this paper, however, requires an intermediate configuration before assembling the target configuration. Sung et al. [184] developed planning methods for the pivoting cube abstraction. Their algorithm executes in $O(N^2)$ time but is restricted to target configurations in the shape of lines.

Unlike lattice-based solution methods, planning algorithms for chain-type systems have to cope with the continuous nature of their motion models. Action sets are not discrete, finite sets but a continuous range of joint angles of modules' actuators. Asadpour et al. [9] approach this problem using a graph-based representation and stochastic optimization methods similar to simulated annealing. A graph similarity metric is used as a heuristic to guide the optimization process in finding a sequence of configurations from an initial to the target configuration. Hou et al. [83] apply graph matching techniques and reduce the reconfiguration problem to a constraint optimization problem. Though specifically developed for chain-type systems, these techniques could also be applied to lattice-based systems.

In this chapter, we present centralized reconfiguration planning approaches that are provably complete, can cope with deadlocks, enclosures, and holes, and feature a worst-case time complexity of $O(N^3)$. Before we present the details of our approach, we review some graph theoretic concepts - most importantly the notion of connectivity. Generally, two grid cells (whether occupied by cubes or empty) are considered to be adjacent if they are located at a distance $d_{L_1} = 1$.[1] This notion of adjacency is used to define the connectivity graph of a configuration as follows.

Definition 1. *Let $G = (V, E)$ be an undirected graph composed of N nodes with $V = \{v_1, v_2, \ldots, v_N\}$, where node v_i represents cube c_i. Then G is called the connectivity graph of*

[1] This notion of adjacency is used in this work because modules are represented by cubes of unit size according to the sliding cube model.



(a) Random configuration.

(b) Corresponding connectivity graph.

Figure 7: Example of a connectivity graph of a random configuration.

a configuration $\mathcal{C}$ *if* $E = V \times V$ *with* $e_{ij} \in E$ *if* $\|c_i - c_j\|_{L_1} = 1$ *(with positions* $c_i, c_j \in \mathcal{C} \subseteq \mathcal{E}$*).*

This definition implies that two nodes v_i, v_j in the connectivity graph are adjacent, if cubes c_i and c_j are located in neighboring grid cells (an example of a configuration and the corresponding connectivity graph is shown in Fig. 7). Note that a connectivity graph can be computed for any set of grid positions, whether these positions are occupied by cubes or not. For example, the *hull* of a configuration (see Def. 4 in Section 3.1.2), which is a set of empty lattice positions can be represented as a connectivity graph. Definition 1 is not only used for reconfiguration planning in this chapter, but also for generating graph grammatical rules in Section 4.1 and for proving completeness and convergence in Section 4.2. We furthermore use the concepts of paths and graph connectivity in the usual graph theoretic sense. Related to graph connectivity is the notion of articulation points or cut vertices. Articulation points become important in Section 3.1.3, where they are used to detect holes and enclosures during reconfiguration.

Definition 2. *An articulation point* v *in a graph* G *is a node whose removal would increase the number of connected components* $c(G)$*, i.e.* $c(G - v) > c(G)$*.*[2]

The notion of connectivity of a configuration allows us to define the constraints we impose

[2] *Connected components of a graph* G *are its maximal connected subgraphs. A connected graph* G *has only one connected component.*

on both homogeneous and heterogeneous systems in this chapter.

Constraints

- *Collision*: A cube c_i is allowed to move to a position $p_i \in \mathcal{E}$ (where $\mathcal{E}$ is a finite environment) if p_i is not already occupied by another cube c_j. Motions are therefore not allowed to cause collisions between cubes.

- *Connectivity*: The configuration $\mathcal{C}$ needs to remain connected at all times. In this sense, a cube requires a connected substrate of other cubes to execute a motion. This constraint also implies that the initial and the target configuration have to be connected.

- *Mobility*: Cubes have to adhere to the motion model outlined in Section 2.3 and are therefore only allowed to perform sliding and corner motions.

- *Permanence*: Once a cube reaches a target position $p_j \in \mathcal{C}_T$, it remains at that target positions until the end of the reconfiguration sequence.

A rigorous definition of permanence requires a notion of system time. For that purpose, let the current configuration at the initial time t_0 be $\mathcal{C}(t_0) = \mathcal{C}_I$. A time step $\triangle t$ is the time required to move a cube $c_i \in \mathcal{C}$ from its initial position to a target position $p_i \in \mathcal{C}_T$. As such, a time step $\triangle t$ is interpreted as the time required for a cube to execute an entire motion sequence. The current time t is therefore defined as $t = t_0 + n\triangle t$, where n is the number of cubes that have been moved to their respective target position. In this sense, the final time t_f is the time at which every cube $c_i \in \mathcal{C}$ has been moved to its respective $p_i \in \mathcal{C}_T$, i.e., $t_f = t_0 + (N-1)\triangle t$, where N is the number of cubes in the configuration $\mathcal{C}$. Note that only $N-1$ cubes have to be moved because of the initial overlap of size one (which will be discussed in Section 3.1.1).

Definition 3. *Permanence requires that once a cube c_i reaches a target position $p_i \in \mathcal{C}_T$ it remains at p_i until $\mathcal{C}_T$ has been fully assembled, i.e., until $\mathcal{C} = \mathcal{C}_T$. More formally, we define*

permanence as

$$c_i(t_{target}) = p_i \in \mathcal{C}_T \quad \rightarrow \quad c_i(t) = p_i \in \mathcal{C}_T,\ \forall t \in [t_{target}, t_f]$$

where t_{target} is the time at which cube c_i first occupied a target position and t_f is the final time.

Permanence therefore requires a cube to remain at the target position p_i that it reached first, which is essential in showing completeness for both the homogeneous planning approach in Section 3.1 as well as the heterogeneous planning approach in Section 3.2.

3.1 Homogeneous Self-Reconfiguration

In this section we present a method for reconfiguring homogeneous configurations. Any initially connected and enclosure-free configuration can be reconfigured into any other connected and enclosure-free target configuration. To focus the contribution of this section on the self-reconfiguration aspect instead of on locomotion, we assume that the initial and target configuration overlap by at least one module. Note however, that multiple pairwise overlapping target configurations can be chained together to enable locomotion as well as the assembly of configurations. The algorithm in this section is provably complete and yields a self-reconfiguration sequence if there exists one (see Theorem 3). Example applications of this homogeneous reconfiguration approach are shown in Section 3.1.5 while numeric results are presented in Section 3.1.6.

3.1.1 Problem Setup

The homogeneous self-reconfiguration problem can be stated in similar terms as the general self-reconfiguration problem in Section 2.2. Given a homogeneous initial and target configuration, find a sequence of primitive module motions that reconfigures the initial into the target configuration. What simplifies the homogeneous compared to the heterogeneous self-reconfiguration problem is the fact that all modules are interchangeable because they are identical in all their properties. The consequence of this interchangeability is that every module can occupy any target position. As stated previously, in this work the sliding cube model is used as an abstraction of self-reconfigurable systems. Therefore module motion

is governed by the motion model outlined in Section 2.3. In addition to the assumptions and constraints of the general sliding cube model, we impose another set of constraints that govern the homogeneous self-reconfiguration planning approach in this section.

Assumptions

- The initial configuration $\mathcal{C}_I$ and the target configuration $\mathcal{C}_T$ are known to the planner.
- $\mathcal{C}_I$ and $\mathcal{C}_T$ contain the same number of modules.
- Both $\mathcal{C}_I$ and $\mathcal{C}_T$ are connected configurations.
- Without loss of generality we assume that there exists an overlap between $\mathcal{C}_I$ and $\mathcal{C}_T$ of exactly one cube c_i which means that c_i is already at its target position. In general, this overlapping region $\mathcal{O} = \mathcal{C}_I \cap \mathcal{C}_T$ can contain more than a single cube as long as it remains connected.
- The configuration is initially enclosure-free and remains enclosure-free throughout the reconfiguration. As shown in Section 3.1.3 (and also in [145]), this assumption is required to ensure that the planning space $\mathcal{N}(\mathcal{C})$ remains connected.
- The target configuration is hole-free. This assumption is required to show completeness of the reconfiguration algorithm in Section 3.1.3. In case the overlap of the initial and the target configuration contains more than one cube, the overlap needs to be hole-free as well to guarantee completeness.

Holes and enclosures are unreachable positions in the target and current configuration, respectively, and are formally defined in Section 3.1.3. They will become important in showing completeness of the reconfiguration algorithm.

3.1.2 Planning Approach

Reconfiguring a homogeneous system requires to move every cube from its initial positions to a target positions. Therefore, a reconfiguration algorithm has to compute paths (sequences of primitive module motions) for cubes $c_i \in \mathcal{C}_I$ to their desired positions $p_i \in \mathcal{C}_T$. Cubes c_i

in the initially overlapping region $\mathcal{O} = \mathcal{C}_I \cap \mathcal{C}_T$ (see Fig. 8a) do not have to be moved and are excluded from the planning process. Reconfiguration planning can be broken down into three basic steps: selecting a mobile cube, assigning a valid target position, and planning a motion sequence for the selected cube.

The connectivity constraint prohibits moving cubes which are articulation points in the connectivity graph of the current configuration because their motion would disconnect the graph and therefore the configuration. Therefore, all articulation points are excluded from the set of mobile cubes, which is furthermore called the *movable set*. The definition of this set depends on the *hull* and the *one-hop-neighborhood* of a set of cubes.

Definition 4. *Given a set of cubes $\mathcal{S}$, its hull $\mathcal{N}(\mathcal{S})$ is the set of all* ***unoccupied*** *lattice positions adjacent to $\mathcal{S}$.*

$$\mathcal{N}(\mathcal{S}) = \left\{ p_i : \min_{c_i \in \mathcal{S}} \|c_i, p_i\|_{L_1} = 1, p_i \in \mathbb{Z}^d \setminus (\mathcal{S} \cup \mathcal{C}) \right\} \tag{1}$$

Here, c_i is a cube in the set $\mathcal{S}$, $\mathcal{C}$ is the current configuration, and p_i is an unoccupied lattice position.

Note that a hull $\mathcal{N}(S)$ can be computed for any set of cubes S (or any set of lattice positions, whether occupied or not). The hull $\mathcal{N}(\mathcal{C})$, in particular, represents all positions adjacent to the current configuration and is used as the planning space for the path planner. It also plays an important role in defining which cubes are mobile at any given time (see Figure 8b) and which positions can be reached from the current configuration $\mathcal{C}$. Similarly, one can define the *k-hop-neighborhood* of a set of lattice positions (occupied or unoccupied) as follows.

Definition 5. *Given a set of lattice positions $\mathcal{S}$, its k-hop-neighborhood $\mathcal{N}_k(\mathcal{S})$ is the set of all* ***occupied*** *lattice positions within a distance of k to $\mathcal{S}$.*

$$\mathcal{N}_k(\mathcal{S}) = \left\{ c_j \in \mathcal{C} : \min_{c_i \in \mathcal{S}} \|c_i, c_j\|_{L_1} \leq k \right\} \tag{2}$$

Here, c_i is a lattice position in the set $\mathcal{S}$ and c_j is an occupied lattice position (or a cube).

The movable set (see Fig. 8b) is furthermore defined as follows, where the set of all articulation points of the connectivity graph $G(\mathcal{C})$ is denoted as $\mathcal{A}$ and the set of cubes

which are *immobile* as I. Note that a cube $c_i \in \mathcal{C}$ is immobile if it is surrounded by four neighbors such that $|\mathcal{N}_1(c_i)| = 4$ (in two dimensions) or six neighbors such that $|\mathcal{N}_1(c_i)| = 6$ (in three dimensions).

Definition 6. *The movable set $\mathcal{M}$ is the set of all cubes at non-target positions that can move without violating collision and connectivity constraints.*

$$\mathcal{M} = \mathcal{C} \setminus (\mathcal{A} \cup I \cup \mathcal{C}_T) \tag{3}$$

This definition (similar to [61]) only allows modules on the surface of the configuration to move. Similarly, cubes at target positions will not be considered mobile and remain fixed for the remaining reconfiguration sequence. Moving a cube from $\mathcal{C}_I$ to $\mathcal{C}_T$ requires an assignment of a mobile cube to an unoccupied target position. Therefore, we introduce the *target successor set* $\mathcal{R}$ as the set of positions $p_i \in \mathcal{C}_T$ that can be reached from the current configuration, i.e. positions adjacent to $\mathcal{C}$ (see Figure 8c). Note that the set of all positions $p_i \in \mathcal{C}_T$ that would create *enclosures* and the set of all positions $p_j \in \mathcal{C}_T$ that would create *holes* if occupied are excluded from the target successor set. These sets are denoted as $\mathscr{E}$ and $\mathscr{H}$ respectively. The concepts of enclosures and holes will be defined rigorously in Section 3.1.3. For now, suffice it to say that holes and enclosures are unreachable positions in the current or target configuration that are completely enclosed by other cubes.

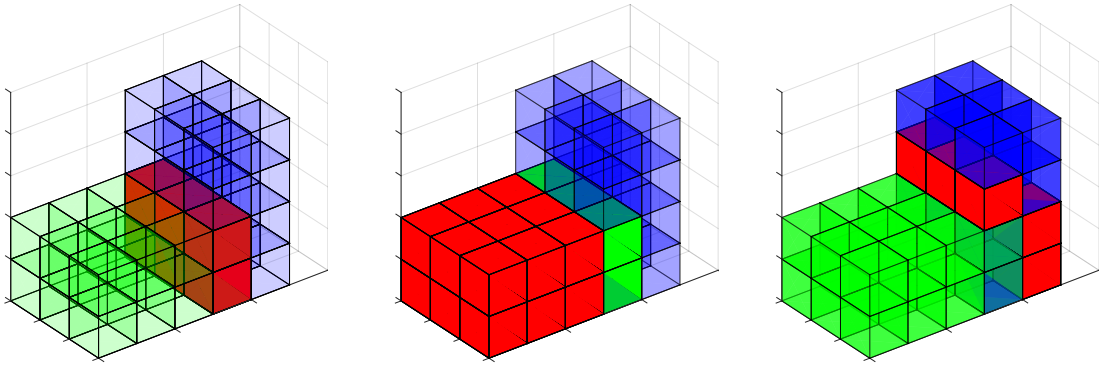
Definition 7. *The target successor set $\mathcal{R}$ is the set of all unoccupied target positions $p_i \in \mathcal{C}_T$ adjacent to currently already occupied target positions $c_i \in \mathcal{C}_T$. Let the set T contain all empty target positions adjacent to $\mathcal{C}$ that do not create holes or enclosures if occupied.*

$$T = \mathcal{C}_T \cap (\mathcal{N}(\mathcal{C}) \setminus (\mathscr{H} \cup \mathscr{E})) \tag{4}$$

Note that T contains all target positions p_i adjacent to $\mathcal{C}$.[3] *This set is further restricted to form $\mathcal{R}$ as follows.*

$$\mathcal{R} = \left\{ p_i \in T : \max_{c_i \in \mathcal{C}_T} \|c_i - p_i\|_{L_1} = 1 \right\} \tag{5}$$

[3] *The connectivity graph of the set T, however, is in general not connected. A disconnected connectivity graph needs to be avoided because otherwise the movable set $\mathcal{M}$ could be empty in certain cases (see case 2 of Lemma 2).*



(a) Initial configuration (translucent green, left), target configuration (translucent blue, right), and overlapping nodes (red, opaque, center)

(b) Movable nodes (red, opaque) as part of the initial configuration (translucent green

(c) Immediate target successor positions (red, opaque) as neighboring positions of the current configuration (translucent green, left)

Figure 8: Representation of the overlapping, the movable, and the immediate target successor set

This definition of $\mathcal{R}$ ensures that all occupied target positions always form a connected connectivity graph, which avoids problematic edge cases such as the one shown in Fig. 10. Note that $\mathcal{R}$ is a subset of the planning space $\mathcal{N}(\mathcal{C})$, in which sequences of primitive cube motions are planned from $c_i \in \mathcal{M}$ to $p_i \in \mathcal{R}$. To ensure completeness of the planning approach, we first have to show that both $\mathcal{M}$ and $\mathcal{R}$ are nonempty unless the target configuration is fully assembled. Proving nonemptiness of $\mathcal{M}$ depends on the following graph theoretic result.

Lemma 1. *According to Lemma 6 in [160], any finite graph $G = (V, E)$ with at least two vertices (such that $|V| \geq 2$) contains at least two vertices that are not articulation points.*[4]

Lemma 2. *The movable set $\mathcal{M}$ is nonempty unless the target configuration $\mathcal{C}_T$ is fully assembled.*

Proof. To prove this lemma, we first note that a movable cube $m_i \in \mathcal{M}$ has to be located on the surface of the configuration because by Def. 6 any $m_i \in \mathcal{M}$ has fewer than four

[4] An articulation point is a vertex in a graph whose removal would disconnect the graph (see Def. 2).

neighbors.[5] Additionally, a movable cube has to be a non-articulation point in the connectivity graph $G_{\mathcal{C}}$ of the current configuration $\mathcal{C}$. We then construct the current configuration inductively with the cubes at target positions being the roots of the configuration. In this construction, one can show that in any such configuration, there exists at least one surface module that is movable. Let $G_T = (V_T, E_T)$ be the connectivity graph of all occupied target positions $c_l \in \mathcal{C}_T$. Then two cases are differentiated. The first case covers all situations for which G_T is connected, while the second case discusses those situations where G_T is disconnected.[6]

Case 1. G_T **is connected.**

This part is proved by induction, where the basic case is as follows. Without loss of generality assume that the vertex set V_T contains a single vertex, which is the case for the initial setup where the overlap of $\mathcal{O} = \mathcal{C}_I \cap \mathcal{C}_T = \{c_l\}$ is a single cube. Furthermore, assume that one cube $c_i \in \mathcal{C} \setminus \mathcal{C}_T$ is connected to c_l. Such a cube is always available unless $\mathcal{C} = \mathcal{C}_T$, i.e. unless the target configuration has been fully assembled. An example of selecting c_i is shown in the first step of Fig. 9. Note that c_i is a surface module by definition (because only two cubes are part of the configuration). Let $G_C = (V_C, E_C)$ be the connectivity graph of the cubes $\{c_i, c_l\}$, which is by definition connected. According to Lemma 1, in any graph, there exist at least two non-articulation point vertices. Because G_C contains only two vertices, both must be non-articulation points. Therefore there exists a surface module that is not an articulation point in G_C and thus movable.

The following induction step shows that adding a cube to the configuration (i.e. adding a vertex to G_C) does not reduce the number of surface modules that are non-articulation points. In other words, the number of movable modules is non-decreasing as new cubes are added. Note that a cube c_j is always added as close as possible to the root cube $c_l \in \mathcal{C}_T$. Two sub-cases have to be differentiated as a new cube c_j is added.[7]

[5] This proof is formulated for two-dimensional reconfiguration, where a cube can have up to four one-hop neighbors. In three-dimensions, the proof follows an equivalent logic with a maximum of six one-hop-neighbors.

[6] Note that a disconnected G_T is ruled out through the construction of the set $\mathcal{R}$. However, it is mentioned here to highlight the issues with a disconnected G_T.

[7] Note that by definition $c_j \in \mathcal{C} \setminus \mathcal{C}_T$.

Case 1.1. **Adding cube c_j removes another module c_k from the movable set.**
A module c_k is removed from the movable set either because it is removed from the set of surface cubes or because it becomes an articulation point. Removal from the surface happens when cube c_j is added as the fourth neighbor of c_k (or the sixth neighbor in three dimensions). This removal decreases the number of surface cubes by one. However, the newly added cube c_j has to be a surface module because of enclosure- and hole-freedom in the current configuration $\mathcal{C}$. In addition, a newly added module cannot be an articulation point in the graph $G_{\mathcal{C}}$ because $G_{\mathcal{C}}$ was already connected before adding c_j. Therefore, the cube c_j is a movable surface module and thus increases the number of movable modules by one. As such the number of movable modules is non-decreasing.

Alternatively, c_k could be removed from $\mathcal{M}$ if it became an articulation point of G_C through the addition of c_j (which is shown in Fig. 9 when cube 5 is added). Since the newly added cube cannot be an articulation point of $G_{\mathcal{C}}$, the number of movable cubes is also non-decreasing.

Case 1.2. **Adding cube c_j does not remove another module c_k from the movable set.**
This is the trivial case, where the newly added cube c_j is both a surface module as well as a non-articulation point in $G_{\mathcal{C}}$ by following the same logic as outlined in the case above (see Fig. 9, for example, when module 4 is added). As such the number of movable cubes in the set $\mathcal{M}$ increases.

Case 2. **G_T is disconnected.**
This case covers scenarios such as the one shown in Fig. 10, where the movable set $\mathcal{M}$ is empty (because all of the cubes $c_i \in \mathcal{C} \setminus \mathcal{C}_T$ are articulation points in $G_{\mathcal{C}}$). The definition of the target successor set $\mathcal{R}$ in Def. 7 rules this case out and only allows the construction of a connected G_T. However, it is instructive to show these edge cases to explain the rational behind the definition of $\mathcal{R}$.

Note that this inductive construction of $\mathcal{C}$ will not prematurely terminate, because the configuration $\mathcal{C}$ is both connected and hole- and enclosure-free. As such, a cube c_i can always

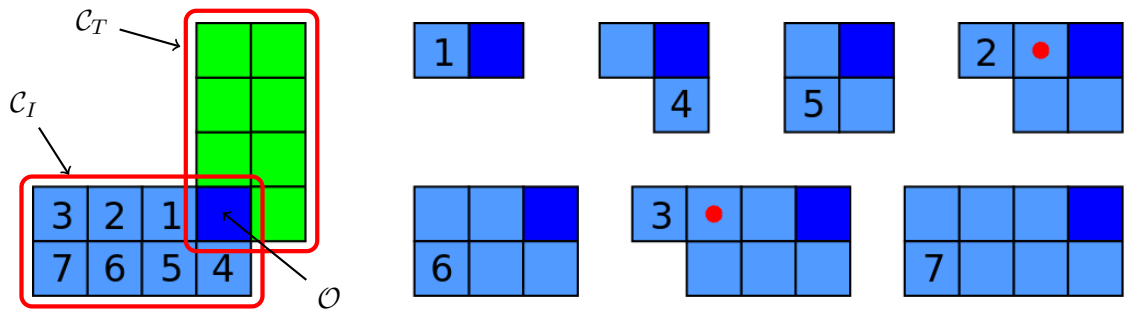


Figure 9: Example construction of a configuration as outlined in case 1 of Lemma 2 to show the nonempty nature of the movable set $\mathcal{M}$. On the left, the initial and target configuration are shown together with the overlap $\mathcal{O} = \mathcal{C}_T \cap \mathcal{C}_T$. On the right, for each step, the currently added cube is numbered while articulation points are marked with a red dot.

be added until the full configuration is constructed. By inductively constructing both the graph $G_\mathcal{C}$ and the current configuration $\mathcal{C}$, we ensure that the number of movable modules is nondecreasing. Furthermore, by showing that a newly added surface module cannot be an articulation point, we have ensured that the movable set $\mathcal{M}$ is nonempty unless $\mathcal{C}_T$ is fully assembled. □

Lemma 3. *The target successor set $\mathcal{R}$ is nonempty until the target configuration $\mathcal{C}_T$ is fully assembled.*

Proof. This proof is based on the observation that the initial configuration $\mathcal{C}_I$ as well as the target configuration $\mathcal{C}_T$ are connected and hole- and enclosure-free by assumption. Additionally $\mathcal{C}_I$ and $\mathcal{C}_T$ are connected through the overlapping set $O = \mathcal{C}_I \cap \mathcal{C}_T$. By maintaining global connectivity throughout the reconfiguration sequence, $\mathcal{C}_I$ and $\mathcal{C}_T$ remain connected at all times.

Let $\mathcal{C}_o$ be the set of all occupied target positions c_i such that $c_i \in \mathcal{C}_T$, $\forall c_i \in \mathcal{C}_o$. Note that the connectivity graph $G_{\mathcal{C}_o}$ of the sub-configuration $\mathcal{C}_o$ is connected by construction, which is guaranteed by the definition of $\mathcal{R}$. Then, the hull $\mathcal{N}(\mathcal{C}_o)$ is nonempty and connected because of hole- and enclosure freedom of $\mathcal{C}_T$. Additionally, the hull of the current configuration $\mathcal{N}(\mathcal{C})$ is also nonempty, connected, and hole- and enclosure-free because $\mathcal{C}$ is. As a consequence, the overlap of $\mathcal{N}(\mathcal{C}_o) \cap \mathcal{N}(\mathcal{C})$ is nonempty because $\mathcal{C}$ and $\mathcal{C}_T$ are connected through the initially overlapping region $O = \mathcal{C}_I \cap \mathcal{C}_T$. The result then follows from the observation that

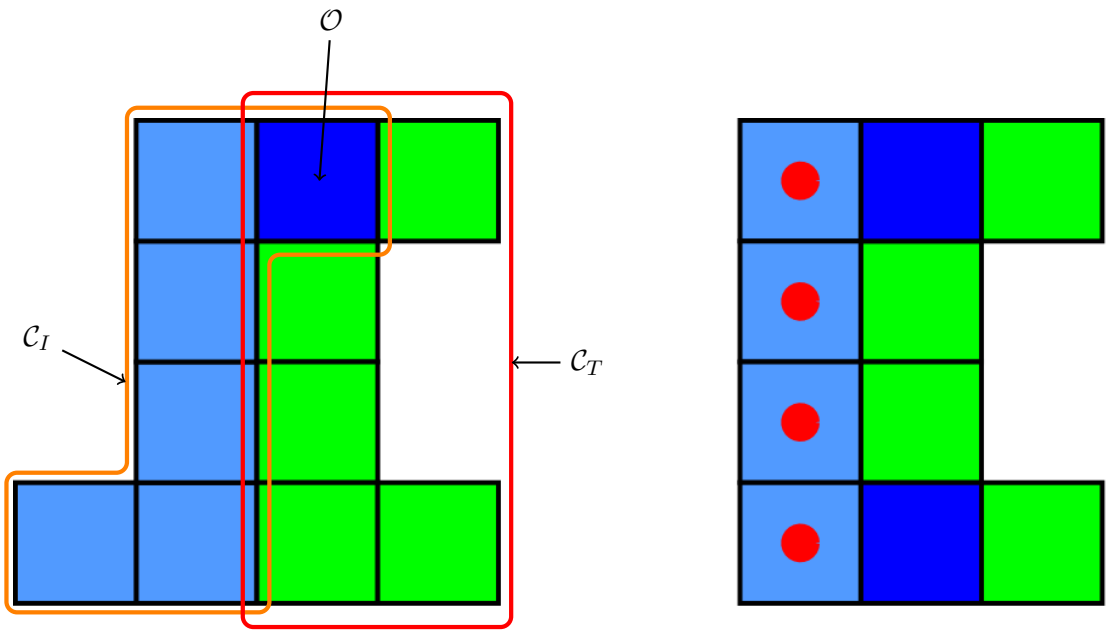


Figure 10: Example of an empty movable set $\mathcal{M}$ according to case 2 of Lemma 2. On the left, the initial and target configuration are shown together with the overlap $\mathcal{O} = \mathcal{C}_T \cap \mathcal{C}_T$. On the right, an empty set $\mathcal{M}$ is shown caused by every cube $c_i \in \mathcal{C}_I \setminus \mathcal{C}_T$ being an articulation point in G_C. Articulation points are marked with a red dot. The definition of $\mathcal{R}$ in Def. 7 specifically rules out a case as shown.

$\mathcal{R}$ can also be expressed as $\mathcal{R} = \mathcal{N}(\mathcal{C}_o) \cap \mathcal{N}(\mathcal{C})$. □

Note that a nonempty set $\mathcal{R}$ contains only those target positions that do not create holes or enclosures if occupied by a cube. Lemma 3, however, hinges on the fact that hole- and enclosure-freedom can be maintained throughout the reconfiguration. This issue will be addressed in Section 3.1.3. Assuming that holes and enclosures can be avoided, Lemma 2 and Lemma 3 show that by construction $\mathcal{M}$ and $\mathcal{R}$ are nonempty unless the target configuration has been assembled. Therefore, there exists a module that can be moved from its initial position to an unoccupied target position at any time (we have shown a similar result in [145]). Given two nonempty sets $\mathcal{M}$ and $\mathcal{R}$, a single movable cube has to be assigned to a single target position before a module path can be planned. In this work, we use a greedy assignment approach to assign a cube $c_i \in \mathcal{M}$ to a target position $p_j \in \mathcal{R}$. The pairwise costs between any cube c_i and target position p_j are computed and the assignment with the lowest cost is chosen for path planning. Paths of primitive agent motions are then planned using a complete planning algorithm. In this work, we use A* with the Manhattan distance as

heuristic function.[8] This assignment and planning step is repeated for all cubes $c_i \in \mathcal{C}_I \setminus \mathcal{C}_T$ and results in a set of paths that describe the entire reconfiguration sequence from $\mathcal{C}_I$ to $\mathcal{C}_T$. The homogeneous self-reconfiguration algorithm is shown in Algorithm 1. While hole and enclosure freedom is required for showing completeness of the homogeneous planning approach, we have neither defined these terms nor described how to detect and avoid them. Hole and enclosure detection and avoidance is therefore the topic of the next section.

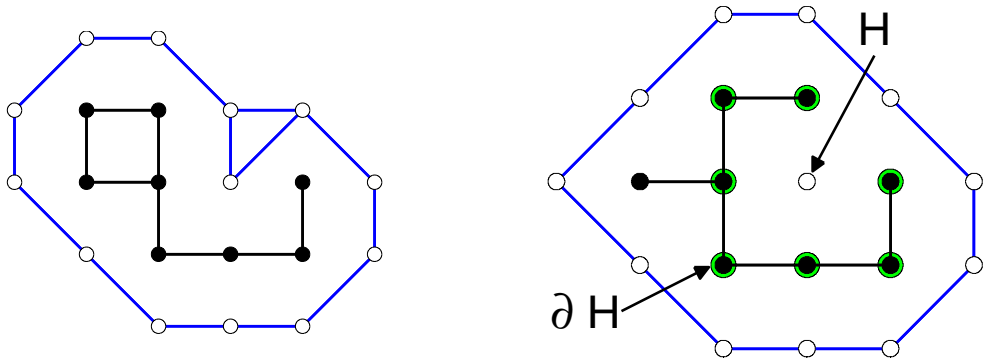
Algorithm 1 Centralized Homogeneous Reconfiguration

Require: Current and target configuration $\mathcal{C}$ and $\mathcal{C}_T$
1: **while** $\mathcal{C} \neq \mathcal{C}_T$ **do**
2: Compute $\mathcal{M}$
3: Compute $\mathcal{R}$
4: Compute assignment $a = \{c_i, p_j\}$, with $c_i \in \mathcal{M}$ and $p_j \in \mathcal{R}$
5: Compute planning space $\mathcal{N}(\mathcal{C})$
6: p = planPath$(a, \mathcal{N}(\mathcal{C}))$
7: executePath(p)
8: **end while**

3.1.3 Hole and Enclosure Detection and Avoidance

The planning approach outlined above potentially creates enclosures or holes in the current configuration, which is an issue because the occurrence of either disconnects the planning space $\mathcal{N}(\mathcal{C})$. However, the completeness of the planning approach in Section 3.1.2 hinges on the connectedness of $\mathcal{N}(\mathcal{C})$. This section will therefore formally introduce the notion of holes as well as methods for detecting and avoiding them. In this research, a hole is defined as a position $h \in \mathcal{C}_T$ that can not be reached by any cube $c_i \in (\mathcal{C} \setminus \mathcal{C}_T)$ at the current time or any future time because of permanence. As we will show, holes obstruct the completion of a reconfiguration sequence because they create permanent deadlocks and must therefore be avoided at all cost. Enclosures are hole-like structures in the current configuration $\mathcal{C} \setminus \mathcal{C}_T$ that present temporary deadlocks because they cannot be reached at the current time. However, because enclosures are not bound by permanence, they may be resolved at a later point in time. Nonetheless, because holes as well as enclosures disconnect

[8] Note that any complete path planning algorithm can be used and the completeness result of Theorem 3 would still hold.



(a) Connectivity graph of a configuration before a hole is created.

(b) Connectivity graph of configuration after a hole is created. ∂H is indicated by green nodes.

Figure 11: Hole detection uses the connectivity graphs of the configuration $\mathcal{C}$ and the planning space $\mathcal{N}(\mathcal{C})$. Note that the current configuration $\mathcal{C}$ is represented by filled black nodes while its hull $\mathcal{N}(\mathcal{C})$ is represented by hollow white nodes.

$\mathcal{N}(\mathcal{C})$ and the detection algorithm (Algorithm 2) relies on a connected planning space, both are prevented from forming. The algorithm presented in this section provably detects holes and can invalidate assignments that would create either. The following definitions formalize holes and their boundaries (also see [71]).

Definition 8. *Let H be a set of unoccupied target positions $p_i \in \mathcal{C}_T \setminus \mathcal{C}$ and ∂H a set of occupied target positions $c_i \in \mathcal{C}_T \cap \mathcal{C}$ such that the following holds.*

$$\partial H = \left\{ \partial h \in \mathcal{C}_T \cap \mathcal{C} : \max_{h \in H} \|\partial h, h\|_{L2} < 2 \right\} \tag{6}$$

Then H is called a hole if for any cube $c_i \in \mathcal{C} \setminus \mathcal{C}_T$ (cubes not at target positions) the following holds.

$$\forall h \in H \; \exists! \; path(c_i, h) \tag{7}$$

In other words, a hole H is an unoccupied target position $c_i \in \mathcal{C}_T$ (or a set thereof) that is surrounded by occupied target positions (the boundary ∂H) such that no path exists between any cube $c_i \in \mathcal{C} \setminus \mathcal{C}_T$ at non-target positions and any position $h \in H$ in the hole. The creation of a hole in a two-dimensional case is shown in Fig. 11, which illustrates that the existence of a hole implies that the planning space $\mathcal{N}(\mathcal{C})$ becomes disconnected and contains more than one connected component (with H being one of them and $\mathcal{N}(\mathcal{C}) \setminus H$

being another). Fig. 11 also shows the boundary of a hole ∂H, which is defined in Eqn. 6 as all occupied target positions at a maximum distance of at most one primitive motion from a position h of hole H. Since both $H \subset \mathcal{C}_T$ and $\partial H \subset \mathcal{C}_T$, once a hole is created, it can not be resolved because of the permanence constraint. Hole-like structures can also occur in the current configuration. These temporary holes are referred to as enclosures and defined as follows.

Definition 9. *Let E be a set of unoccupied positions $p_i \in \mathcal{E}$ (where $\mathcal{E}$ is the environment) and ∂E a set of occupied positions $c_i \in \mathcal{C}$ such that the following holds.*

$$\partial E = \left\{ \partial e \in \mathcal{C} : \max_{e \in E} \|\partial e, e\|_{L2} < 2 \right\} \tag{8}$$

Then E is called an enclosure if for any cube $c_i \in \mathcal{C} \setminus \partial E$ (cubes not in the boundary of the enclosure) the following holds.

$$\forall e \in E \; \exists! \; path(c_i, e) \tag{9}$$

Note that a hole (and its boundary) are entirely contained within the target configuration. An enclosure may be partly contained within $\mathcal{C}_T$ or lie entirely outside $\mathcal{C}_T$. Therefore, permanence does not apply to enclosures and they do not obstruct the successful completion of a reconfiguration sequence. However, enclosures disconnect the planning space $\mathcal{N}(\mathcal{C})$, which is why assignments that create enclosures are also invalidated. The importance of hole avoidance is shown in the following theorem.

Theorem 1. *A hole obstructs the successful completion of a reconfiguration sequence.*

Proof. The proof follows directly from the definition of a hole, where $H \in \mathcal{C}_T$, i.e. H contains unoccupied target positions. By definition, $\partial H \in \mathcal{C}_T$, which means that because of the permanence constraint none of the cubes $c_i \in \partial H$ will be moved until the reconfiguration process is completed. However, the reconfiguration process will never terminate because it contains a hole H and its boundary ∂H, which blocks the occupation of any empty position $h \in H \subset \mathcal{C}_T$. Therefore, a reconfiguration sequence cannot be completed when a hole is exists. □

Theorem 1 shows that a successful reconfiguration sequence cannot contain any holes. The following algorithm (Algorithm 2) provably detects both holes and enclosures and enables the avoidance of either. In a nutshell, Algorithm 2 determines whether a given assignment disconnects the planning space $\mathcal{N}(\mathcal{C})$. It is based on the assumption of an initially connected planning space $\mathcal{N}(\mathcal{C})$, which is the case because $\mathcal{C}_I$ is connected and hole- and enclosure-free. In a first step, Algorithm 2 computes the number of connected components of the connectivity graph $G_{\mathcal{N}(\mathcal{C})}$ of the planning space $\mathcal{N}(\mathcal{C})$, which is equal to the number of zero eigenvalues of the associated graph Laplacian L (according to [123]).

If $G_{\mathcal{N}(\mathcal{C})}$ is disconnected, the algorithm reports the detection of a hole or an enclosure for any assignment and will essentially stop the reconfiguration. Therefore, it is crucial to maintain hole- and enclosure-freedom throughout the reconfiguration. If $G_{\mathcal{N}(\mathcal{C})}$ is connected, Algorithm 2 determines whether the execution of given assignment $a = \{c_i, p_j\}$ creates a hole or an enclosure. A hypothetic move of cube c_i to position p_j is executed (lines 7 to 8) and the number of connected components is recomputed (lines 9 to 10). An increase in the number of zero eigenvalue indicates a disconnected planning space, which invalidates assignment a. Furthermore, Def. 8 and Def. 9 can be used to differentiate between holes and enclosures. However, since holes as well as enclosures have to be avoided, Algorithm 2 simply determines whether a given assignment would disconnect $\mathcal{N}(\mathcal{C})$.

Algorithm 2 Hole Detection

Require: Assignment $a = \{c_i, p_j\}$ and current configuration $\mathcal{C}$
1: Compute $\mathcal{N}(\mathcal{C})$
2: Compute $G_{\mathcal{N}(\mathcal{C})}$ of $\mathcal{N}(\mathcal{C})$
3: Compute L of $G_{\mathcal{N}(\mathcal{C})}$
4: **if** $|\lambda_i = 0| > 1, \forall \lambda \in eig(L)$ **then**
5: Return true
6: **else**
7: Remove p_j from $\mathcal{N}(\mathcal{C})$
8: Update c_i's origin to p_j (in $\mathcal{C}$)
9: Recompute $\mathcal{N}(\mathcal{C})$, $G_{\mathcal{N}(\mathcal{C})}$, and L
10: **if** $|\lambda_i = 0| > 1, \forall \lambda \in eig(L)$ **then**
11: Return true
12: **else**
13: Return false
14: **end if**
15: **end if**

As long as $\mathcal{C}_I$ is a hole- and enclosure-free initial configuration, the reconfiguration algorithm (Algorithm 1) will guarantee hole and enclosure freedom throughout the reconfiguration sequence because Algorithm 2 will detect any holes and enclosures and invalidate assignments that create either. The following theorem shows that this algorithm indeed guarantees the detection of holes and enclosures.

Theorem 2. *Algorithm 2 detects a hole (or an enclosure) if and only if there exists a hole (or enclosure).*

Proof. The proof is based on the assumption that the current configuration $\mathcal{C}$ is hole and enclosure-free. Therefore, the connectivity graph $G_{\mathcal{N}(\mathcal{C})}$ of $\mathcal{N}(\mathcal{C})$ is connected.

- Necessity (D $\rightarrow$ H): The detection of a hole is based on the eigenvalues λ_i of the graph Laplacian L, where the multiplicity of $\lambda_i = 0$ (the number of zero eigenvalues) indicates the number of connected components of the graph. Therefore, we can conclude that $G_{\mathcal{N}(\mathcal{C})}$ is disconnected if the multiplicity of $\lambda_i = 0$ is larger than 1. By construction, $\mathcal{N}$ is connected, which means that the occurrence of two or more connected components implies the existence of either a hole or an enclosure.

- Sufficiency (H $\rightarrow$ D): Starting with an initially connected $\mathcal{N}$ and using the fact that a hole or an enclosure increases the number of connected components of $G_{\mathcal{N}(\mathcal{C})}$, the hole-detection problem is reduced to determining how many connected components $G_{\mathcal{N}(\mathcal{C})}$ contains. If the configuration $\mathcal{C}$ contains a hole or an enclosure, L has more than one zero eigenvalue, and we immediately detect the hole or the enclosure.

□

3.1.4 Completeness

One important property of any self-reconfiguration algorithm is the guarantee that it terminates successfully and provably yields a reconfiguration sequence from the initial configuration to the target configuration. According to [161], a complete algorithm is one that is guaranteed to find a solution if there exists one. Using the hole detection algorithm from Section 3.1.3 we can now prove this completeness property.

Theorem 3. *Given any pair of hole- and enclosure-free two- or three-dimensional initial and target configurations $\mathcal{C}_I$ and $\mathcal{C}_T$, the homogeneous reconfiguration algorithm (Algorithm 1) will compute a sequence of primitive cube motions that reconfigure $\mathcal{C}_I$ into $\mathcal{C}_T$ if such a sequence exists.*

Proof. Lemma 2 and Lemma 3 guarantee that both the movable set $\mathcal{M}$ and the reachable set $\mathcal{R}$ are nonempty unless $\mathcal{C}_T$ is fully assembled. Therefore, at every time step, an assignment $a = (c_i, p_i)$ with $c_i \in \mathcal{M}$ and $p_i \in \mathcal{R}$ exists. This statement holds because in a homogeneous system, any movable cube $c_i \in \mathcal{M}$ can be assigned and moved to any target position $p_i \in \mathcal{R}$.

A valid assignment guarantees that a path can be planned from c_i to p_i through $\mathcal{N}(\mathcal{C})$ because $\mathcal{N}(\mathcal{C})$ is connected by construction. The connectivity of $\mathcal{N}(\mathcal{C})$ is guaranteed by Theorem 2, which ensures enclosure and hole freedom of the current configuration $\mathcal{C}$. Because a complete path planner is used for planning individual cube paths, such a path will be found because it is guaranteed to exist (since $\mathcal{N}(\mathcal{C})$ is connected and the target position $p_i \in \mathcal{N}(\mathcal{C})$).

Every valid assignment enables a cube to be moved to the target configuration $\mathcal{C}_T$; and, therefore, the number of cubes occupying target positions $p_j \in \mathcal{N}(\mathcal{C})$ is strictly monotonically increasing in time. Because the configuration consists of a finite number of cubes, this process will terminate in a finite number of steps and $\mathcal{C}_T$ is guaranteed to be assembled.[9] □

3.1.5 Examples

This section shows a number of example applications of the centralized homogeneous self-reconfiguration approach presented above. These examples include locomotion through self-reconfiguration, self-assembly starting from an initial random configuration, reconfiguration from a structured initial configuration to another, as well as obstacle-constrained self-reconfiguration.

Locomotion In this example, a cubic initial configuration is translated along the x-axis by 1.5 times the length of the initial configuration. As can be seen in Fig. 12, the shown

[9] A similar proof of completeness was also shown in [145].

initial and target configurations do not overlap, which is why an intermediate configuration (not shown in the figure) is required. The cubic initial configuration is reconfigured to an intermediate cubic configuration that overlaps both the initial and the final target configuration.

Assembly This example shows the capabilities of Algorithm 1 for the purposes of self-assembly. In the specific case shown in Fig. 13, a random two-dimensional initial configuration is reconfigured into a three-dimensional chair configuration. One can recognize chain-like branches extending from the base of the chair configuration, which is an artifact of the greedy assignment method as well as connectivity maintenance throughout the reconfiguration sequence.

Reconfiguration This example shows a reconfiguration sequence from a structured initial configuration to another structured target configuration. In this particular setup in Fig. 14, a chair is reconfigured into a table configuration. Note that here, the initial overlapping region $O = \mathcal{C}_I \cap \mathcal{C}_T$ contains more than a single cube, which the assumptions shown in Section 3.1.1 allow and Theorem 3 covers.

Obstacle-constrained Reconfiguration Similar to the above reconfiguration example, this last scenario in Fig. 15 shows an obstacle-constrained self-reconfiguration sequence from an initial chair to a table configuration. In this case, the planning space $\mathcal{N}(\mathcal{C})$ is restricted by a set of obstacle positions which represent a ground plane. In other words, the reconfiguration is restricted to the lattice positions above the ground plane. Note, however, that all the theoretic results in this section as well as the completeness theorem (Theorem 3) still hold.

3.1.6 Results

This section presents simulation results obtained in two experiments. We simulated the reconfiguration of overlapping configurations in the form of rectangular prisms (see Fig. 16) and the reconfiguration of overlapping random configurations (see Fig. 17). These

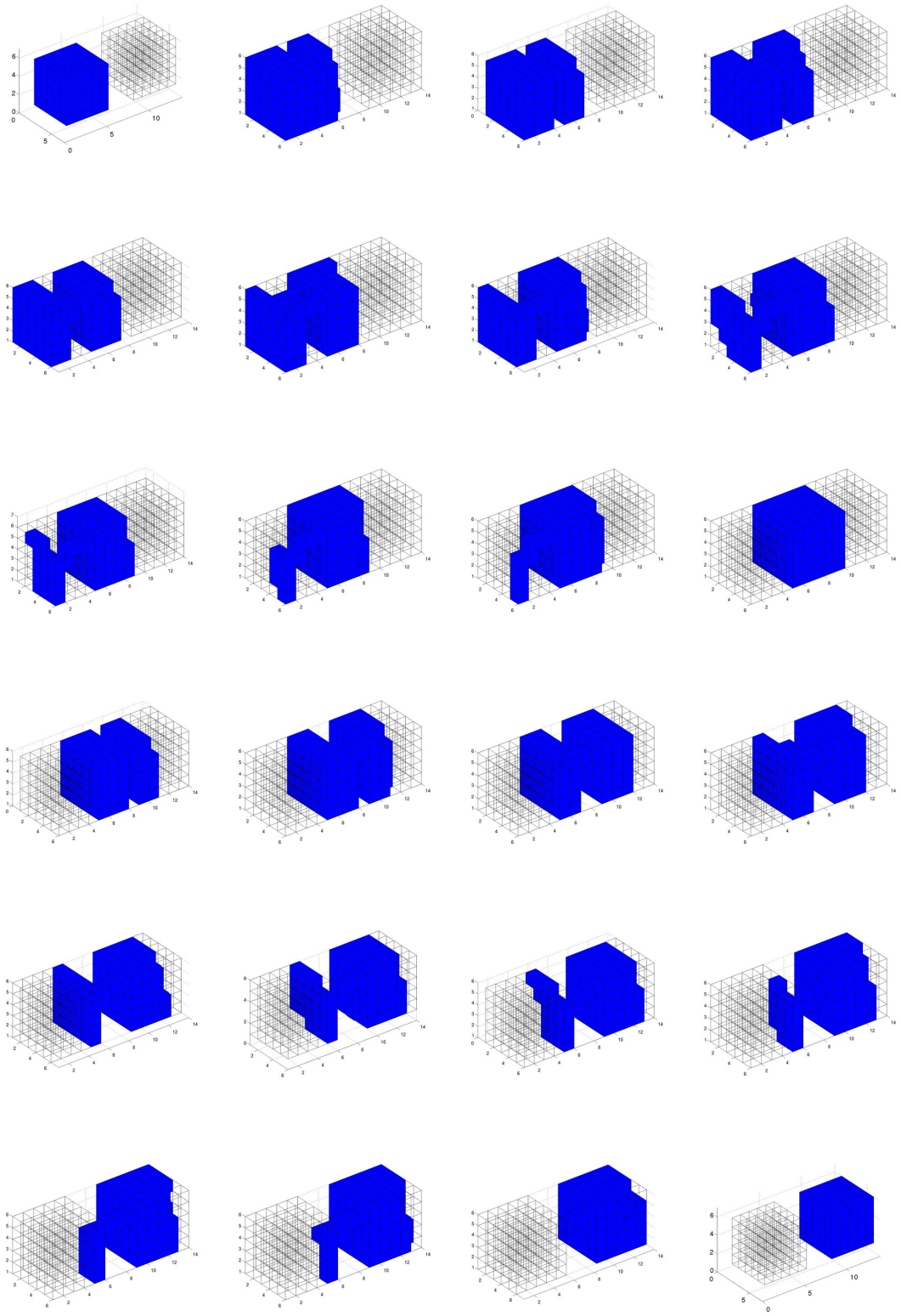


Figure 12: Example of a homogeneous reconfiguration sequence used for locomotion.

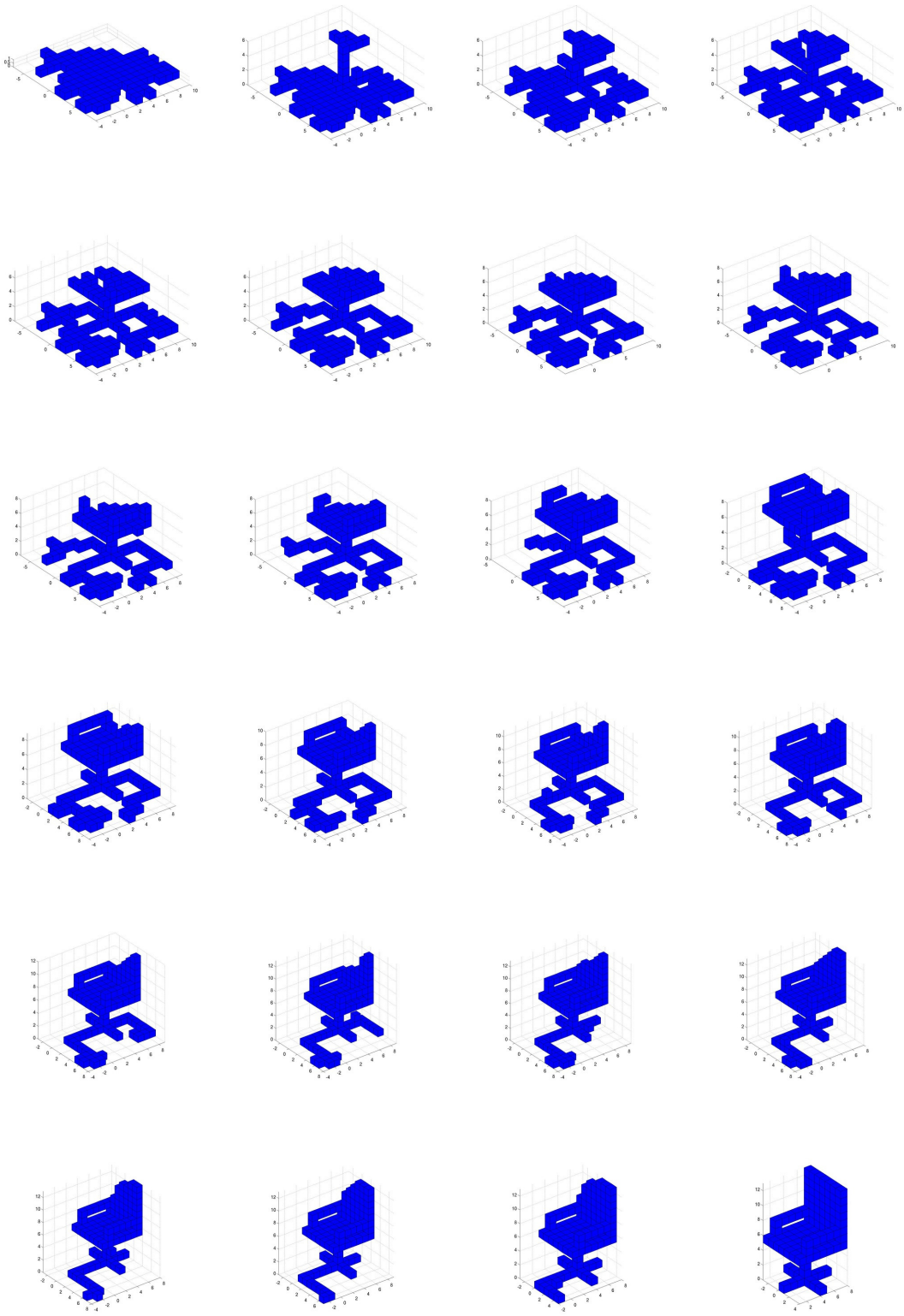


Figure 13: Example of a homogeneous reconfiguration sequence showing the assembly of a chair configuration given a random initial configuration.

Figure 14: Example of a homogeneous reconfiguration sequence showing a reconfiguration from a chair to a table configuration.

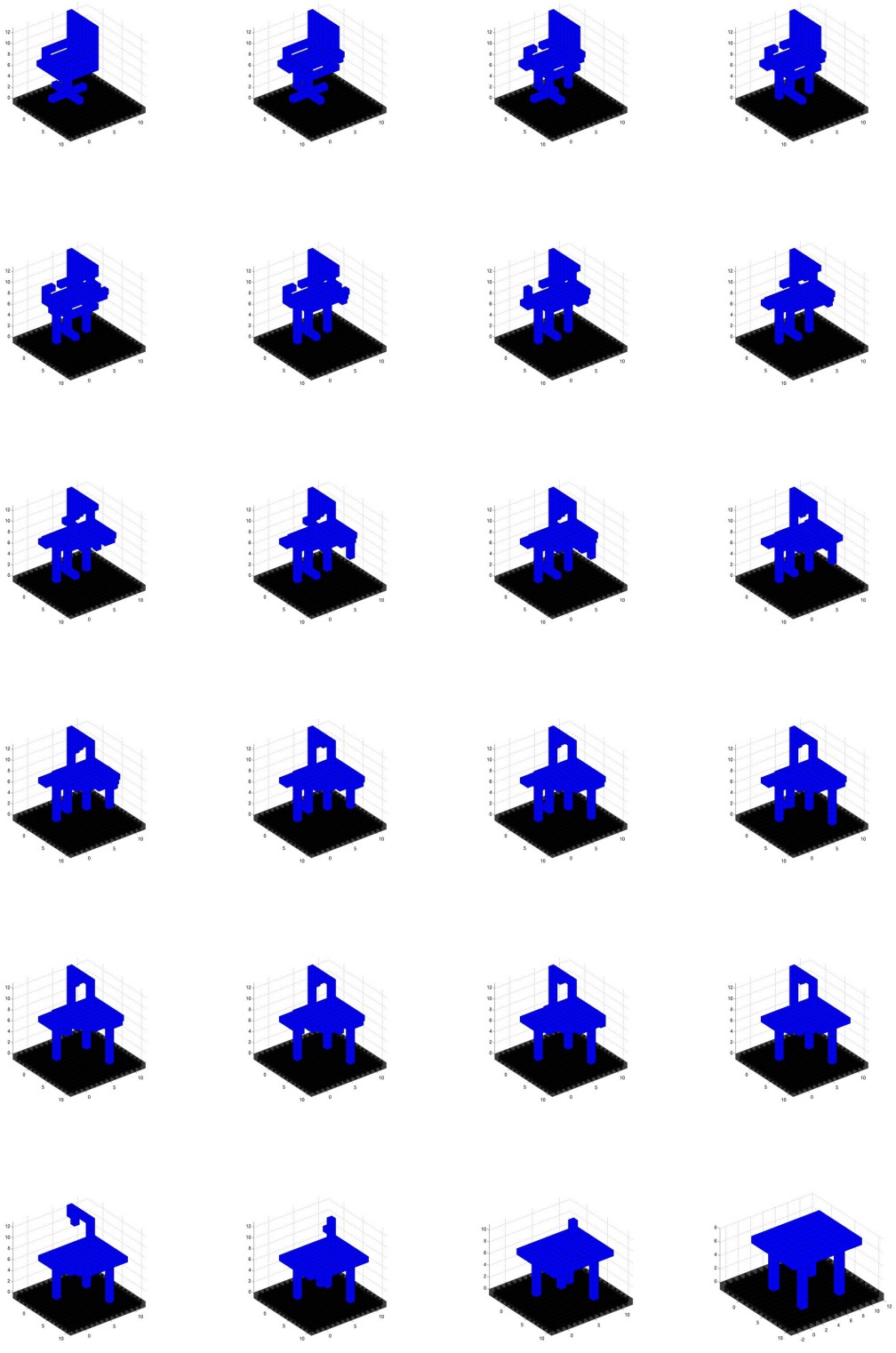


Figure 15: Example of a homogeneous reconfiguration sequence showing a reconfiguration from a chair to a table configuration with obstacles in the environment. In this specific example, obstacles are shown as black cubes and represent a ground plane.

Table 1: Reconfiguration planning results for overlapping box configurations

Size	**Overlap [N]/[%]**	**Path Length**	**Runtime [min]**
100	30 / 30%	837	3.70
200	60 / 30%	1543	16.65
300	90 / 30%	2426	63.26
400	120 / 30%	3279	135.64
500	150 / 30%	4275	246.93

configurations ranged in size from 100 to 500 modules. Our test system was equipped with an Intel Core i5-540M dual core processor and 4GB of DDR3 memory. Our simulator was implemented in Matlab 2010a running on Ubuntu 11.04. As shown in the figures, the cumulative path length increases approximately linearly with the number of modules in the system, while the runtime of the centralized homogeneous algorithm (Algorithm 3) increases approximately with the third power of N for the box and random configurations. The runtime is primarily determined by the planning approach (introduced in Section 3.1.2), which necessitates planning a path for every individual module. Algorithm 3.1.2 features a time complexity of $O(N^2)$ for the relocation of an individual cube and a total time complexity of $O(N^3)$ for the entire reconfiguration process. The experimental results shown in Fig. 16 and Fig. 17 confirm this expected time complexity of $O(N^3)$. The details of each reconfiguration sequence for all sizes are summarized in Table 1 and Table 2. In these tables, the field *Size* refers to the number of modules in the configuration, *Overlap* is the number of initially overlapping modules, *Path Length* is the cumulative path length of all cubes to achieve the desired reconfiguration, and *Runtime* is the time it took to complete the planning stage. Note that unlike in the assumptions outlined in Section 3.1, the overlapping region contains more than a single cube. Such overlapping regions are possible as long as they do not contain any enclosures or holes. The configurations used for the results shown below satisfy these requirements.

Table 2: Reconfiguration planning results for overlapping random configurations

Size	Overlap [N]/[%]	Path Length	Runtime [min]
100	36 / 36%	352	2.25
200	114 / 57%	403	10.97
300	157 / 52.3%	893	40.52
400	182 / 45.5%	1674	120.84
500	231 / 46.2%	2327	272.68

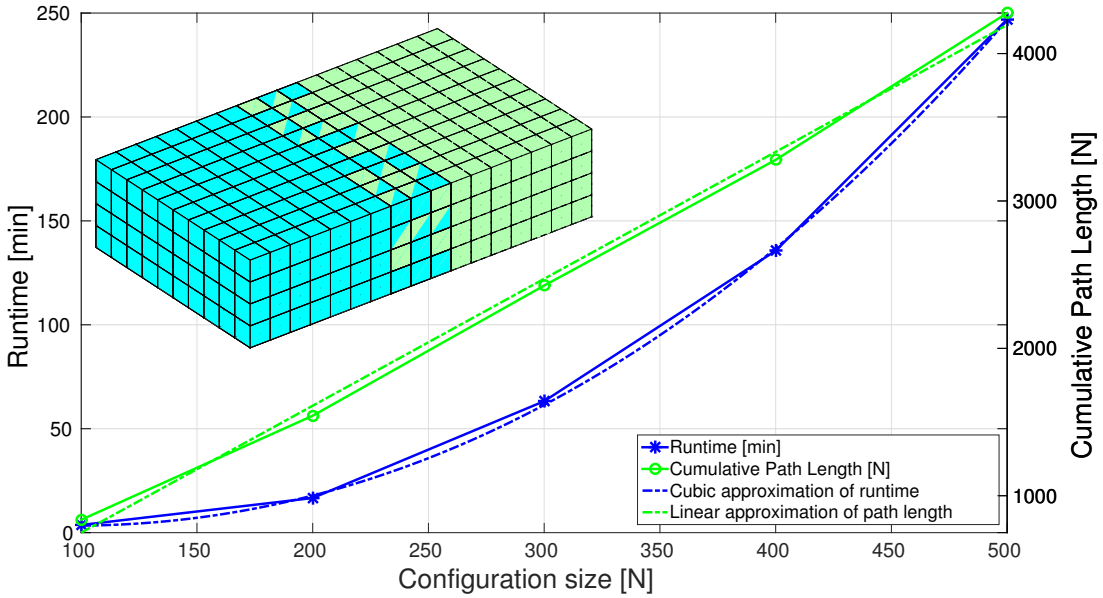


Figure 16: Cumulative length of paths and required runtime of reconfiguration of box configurations.

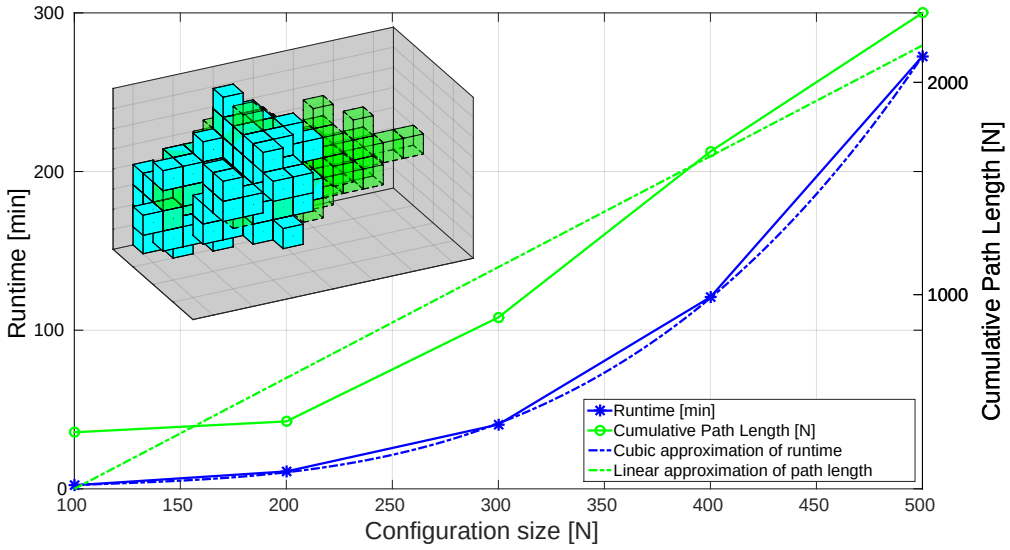


Figure 17: Cumulative length of paths and required runtime of reconfiguration of random configurations.

3.2 Heterogeneous Self-Reconfiguration

The existing self-reconfiguration literature focused extensively on homogeneous systems, in which all modules are identical and interchangeable (for example [160, 189, 101, 103, 73, 30]). Whereas homogeneity typically reduces the complexity of reconfiguration algorithms (see [11]), homogeneous reconfiguration can not guarantee absolute module positioning in the sense that a module c_i is guaranteed to be moved to a specific target position p_i. It is simply moved to any location in the target configuration (see [62]). Since modules are interchangeable, this restriction does not matter in a homogeneous system. Heterogeneous systems on the other hand are comprised of modules with different capabilities and potentially shapes and sizes. In this case, interchangeability of modules is lost and it is conceivable that absolute placement of modules becomes important. Therefore, heterogeneous reconfiguration algorithms (for example [61, 65, 62]) have to be able to move a module c_i to a suitable target position p_i. For example, a heterogeneous reconfiguration algorithm has to guarantee that any target position suitable only for battery modules is occupied by no other type than battery modules.

Another restriction of homogeneous modular robots is their limited extendability of

functionality (see [11, 62]). One can of course add more modules to a homogeneous robot, but extending module capabilities is difficult because either all modules have to be extended by the same capability (e.g. an additional sensor) or the basic assumption of homogeneity is violated. As soon as a single module differs from all others, the robot has to be treated as a heterogeneous system. The relevant characteristics of heterogeneous systems in this section are summarized as follows:

- Heterogeneous robots are comprised of (groups of) modules with distinct properties.
- Without loss of generality, in this section, modules differ only in color as the defining characteristic.
- All modules are identical in size and shape.
- Absolute positioning of modules needs to be guaranteed for heterogeneous robots.[10]

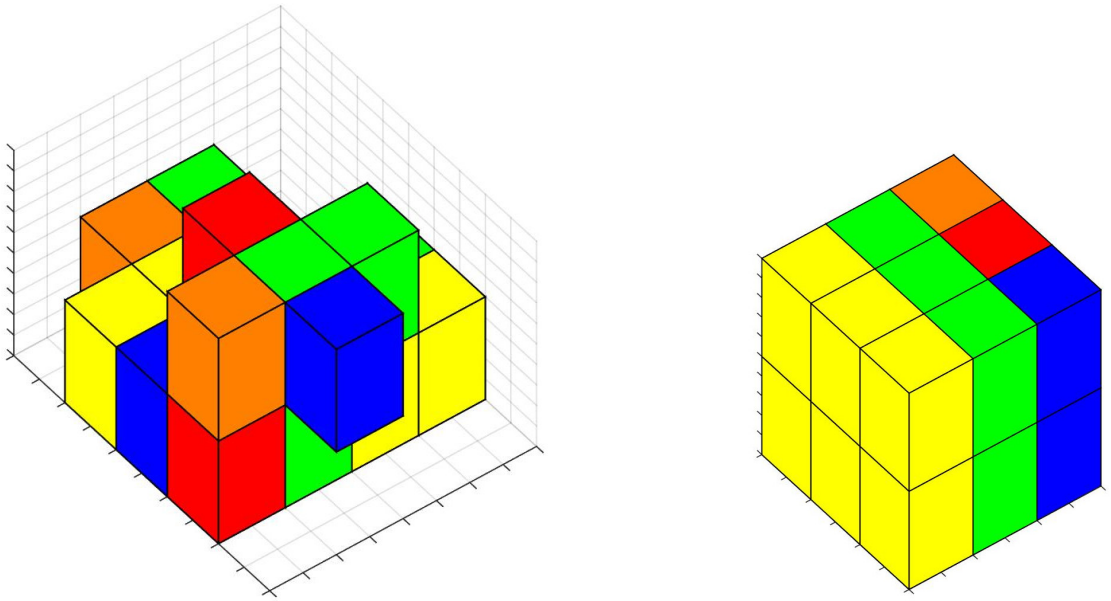
The loss of interchangeability in heterogeneous systems introduces additional complications such as the potential for creating deadlocks (see [28]). In the presented algorithm, the avoidance of deadlocks translates into the avoidance of holes and enclosures (similarly to the homogeneous case) during the reconfiguration process. Additionally, we have to ensure the existence of valid assignments to avoid deadlocks.

This section outlines an approach for reconfiguring heterogeneous systems from any connected, initially hole- and enclosure-free configuration into any other connected, hole- and enclosure-free target configuration. The main theoretic contribution is a provably complete algorithm for heterogeneous systems that avoids deadlocks. The planning approach builds on Section 3.1 and previous work we have published in [145, 146].

3.2.1 Problem Setup

The heterogeneous self-reconfiguration problem is equivalent to the homogeneous problem described in Section 3.1.1 with the additional twist that a module has to be assigned to a

[10] If a configuration contains groups of modules with identical properties, these modules are interchangeable within their group. Within a group, identical modules act like a homogeneous system. Absolute module placement can then only guarantee that a module is placed at any target position suitable for that group.



(a) (b)

Figure 18: Example of a heterogeneous initial and target configuration comprised of colored unit cubes.

target position with matching properties. As such, modules are no longer interchangeable and the assignment of modules to target positions has to take this loss of interchangeability into account. The solution to the heterogeneous self-reconfiguration problem is still a set of individual module paths that move all modules from their initial positions to their respective and matching target positions.

A heterogeneous configuration is once again represented using the sliding cube model detailed in Section 2.3 but the state of a cube now contains its origin, its ID, as well as a set of properties. Without loss of generality, we represent the heterogeneity of cubes with different colors (see Fig. 18). Therefore the set of properties of cubes contains a single element - color. This restriction can be made because even if cubes differ in just one property the assumption of homogeneity is violated and the system has to be treated as a heterogeneous system. The full set of assumptions imposed on the heterogeneous system is summarized below.

Assumption

- The initial configuration $\mathcal{C}_I$ and the target configuration $\mathcal{C}_T$ are known.
- $\mathcal{C}_I$ and $\mathcal{C}_T$ contain the same number of modules of each property.
- Both $\mathcal{C}_I$ and $\mathcal{C}_T$ are connected configurations.
- Without loss of generality we assume that there exists an overlap between $\mathcal{C}_I$ and $\mathcal{C}_T$ of exactly one cube. This cube matches the properties of its position (i.e., it is already at its target position).
- The configuration is initially enclosure-free and remains enclosure-free throughout the reconfiguration. As shown in Section 3.2.2 (and also in [145, 146]), this assumption is required to ensure that the planning space $\mathcal{N}(\mathcal{C})$ remains connected.
- The overlap of the initial and the target configuration is hole-free. This assumption is required to show completeness of the reconfiguration algorithm in Section 3.1.3.

Holes and enclosures have been defined in Section 3.1.3 and will become important in showing completeness of the heterogeneous reconfiguration algorithm in Section 3.2.4.

3.2.2 Planning Approach

Heterogeneous reconfiguration planning requires the computation of motion sequences for all cubes from their position c_i in the initial configuration $\mathcal{C}_I$ to matching positions p_i in the target configuration $\mathcal{C}_T$. The heterogeneous planning approach is similar to the homogeneous method outlined in Section 3.1.2 in the sense that as a first step, the movable set of cubes $\mathcal{M}$ (see Def. 6), the planning space $\mathcal{N}(\mathcal{C})$ (see Def. 4), as well as the target successor set target successor set $\mathcal{R}$ (see Def. 7) are computed. As before, global connectivity has to be maintained throughout the reconfiguration sequence, which means that articulation points of the connectivity graph $G_{\mathcal{C}}$ are excluded from movable set $\mathcal{M}$. In addition, all cubes $c_i \in \mathcal{C}_T$ that already occupy target positions are considered immobile in the sense of permanence (see Def. 3).

The main difference between heterogeneous and homogeneous reconfiguration planning

is that target positions $p_i \in \mathcal{R}$ in the target successor set of a heterogeneous target configuration have properties just like actual cubes $c_i \in \mathcal{C}$.[11] Whereas for homogeneous systems, any cube in $c_i \in \mathcal{M}$ can be moved to any target position $p_i \in \mathcal{R}$ (due to interchangeability), a heterogeneous assignment requires matching properties of $c_i \in \mathcal{M}$ and $p_i \in \mathcal{R}$, which gives rise to the notion of valid assignments.

Definition 10. *Let Q be the set of all properties defined for cubes. A valid assignment $a = \{c_i, p_i\}$ with $c_i \in \mathcal{M}$ and $p_i \in \mathcal{R}$ is one where all properties of c_i and p_i match. More formally it is defined as follows.*

$$a = \{c_i, p_i\}, \textit{ with } c_i \in \mathcal{M}, p_i \in \mathcal{R}$$
$$\textit{and } q_k(c_i) = q_k(p_i), \forall q_k \in Q$$

As shown in Lemma 2 and Lemma 3, by construction $\mathcal{M}$ and $\mathcal{R}$ are nonempty sets unless the target configuration has been assembled. For homogeneous reconfiguration, there exists a module that can be moved from its initial to a target position at any time. For heterogeneous self-reconfiguration, this is not generally true even for nonempty sets $\mathcal{M}$ and $\mathcal{R}$. If no cube $c_i \in \mathcal{M}$ matches the properties of any $p_i \in \mathcal{R}$ then no valid assignment can be found. An example is shown in Fig. 19 where $\mathcal{M}$ contains only red cubes and $\mathcal{R}$ only yellow ones. To avoid deadlocks created by a lack of valid assignments, an algorithm called assignment resolution is used (see Algorithm 3, line 7). After a valid assignment has been computed, the algorithm plans a path from $c_i \in \mathcal{M}$ to $p_i \in \mathcal{R}$ through planning space $\mathcal{N}(\mathcal{C})$ equivalently to the homogeneous reconfiguration planning approach in Section 3.1.2. The complete algorithm is outlined below as Algorithm 3.

3.2.3 Assignment Resolution

The purpose of assignment resolution is to resolve a lack of valid assignments and avoid deadlocks that would prevent the successful completion of the heterogeneous reconfiguration sequence. As such, assignment resolution is only needed if no valid assignment according to Def. 10 can be found. In that case a randomly chosen mobile cube is moved to a temporary

[11] Empty positions $p_i \in \mathcal{N}(\mathcal{C})$, however, do not have any properties, i.e. any module can occupy a position in $\mathcal{N}(\mathcal{C})$, which is essential for path planning.

Algorithm 3 Centralized Heterogeneous Reconfiguration

Require: Current and target configuration $\mathcal{C}$ and $\mathcal{C}_T$

while $\mathcal{C} \neq \mathcal{C}_T$ **do**
  Compute $\mathcal{M}$
  Compute $\mathcal{R}$
  Compute planning space $\mathcal{N}(\mathcal{C})$
  Compute assignment $a = \{c_i, p_j\}$
  while !isValid(a) **do**
    assignmentResolution($\mathcal{M}$, $\mathcal{R}$, $\mathcal{N}(\mathcal{C})$, E)
    Recompute $\mathcal{M}$
    Recompute assignment $a = \{c_i, p_j\}$
  end while
  p = planPath(a, $\mathcal{N}(\mathcal{C})$)
  executePath(p)
end while

position in the hull $\mathcal{N}(\mathcal{C})$ of the current configuration according to the following assignment resolution rule (as in Section 3.1.2, the set of positions in $p_i \in \mathcal{N}(\mathcal{C})$ that would create enclosures if occupied is denoted as $\mathscr{E}$).

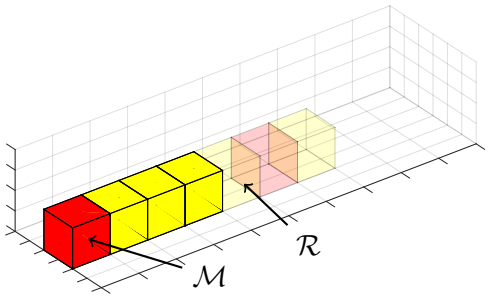
Definition 11. *Assignment resolution computes an assignment as follows.*

$$a = \{m_i, t_i\}$$

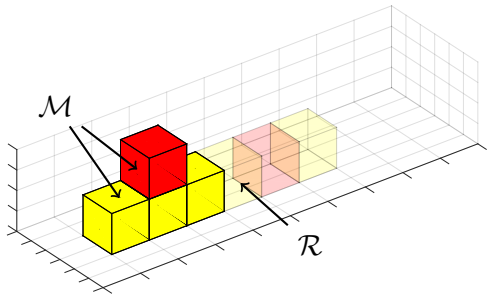
$$\text{with } t_i = \text{rand}(\mathcal{N}(\mathcal{C}) \setminus (\mathcal{R} \cup \mathcal{N}(\mathcal{R}) \cup \mathscr{E}))$$

Instead of moving a cube $m_i \in \mathcal{M}$ to a target position, it is moved to a random temporary position t_i in the hull of the current configuration which is neither a reachable target position nor in the neighborhood of a reachable target position. Additionally, m_i can not be picked such that its occupation will create an enclosure.

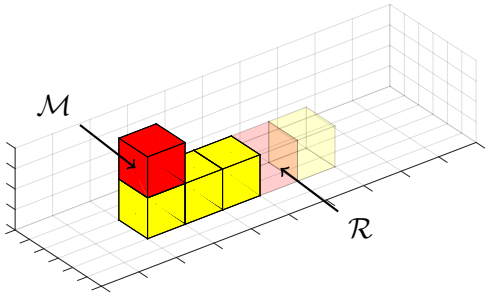
The rationale behind assignment resolution is that as long as cubes $m_i \in \mathcal{M}$ are moved randomly to positions assigned by Def. 11, assignment resolution will eventually add a cube to the movable set whose properties match those of a position $p_i \in \mathcal{R}$. At that point, a valid assignment can be computed and the assembly of the target configuration can continue. Note that assignment resolution will never obstruct the assembly of $\mathcal{C}_T$ because it will never move a cube to a reachable target position nor into the neighborhood of a reachable target position. Therefore, it will never create a hole or an enclosure. Another consequence of assignment resolution is that a target position will never be occupied by a non-matching



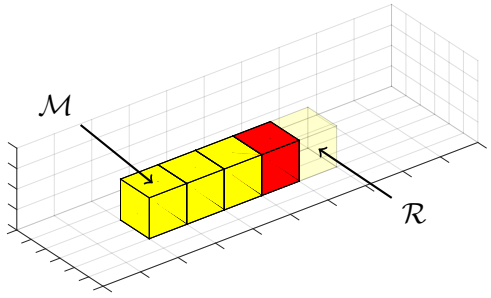
(a) Step 1: No valid assignment



(b) Step 2: Result of random motion of the red cube



(c) Step 3: Valid assignment results in yellow cube being moved to $p_i \in \mathcal{C}_T$



(d) Step 3: Valid assignment results in red cube being moved to $p_i \in \mathcal{C}_T$

Figure 19: Example of a reconfiguration sequence using assignment resolution. Opaque cubes represent the current configuration, transparent cubes represent unoccupied target positions. The goal is to move the line configuration three steps to the right. Shown are two assignment resolution step, in which the red cube has to be moved two steps to the right before the reconfiguration can continue.

cube.

Theorem 4. *Assignment resolution will enable the computation of a valid assignment with probability 1.*

Proof. This result is shown in two steps. First we show that a deterministic sequence of moves can reconfigure all cubes that are not at their target position into an intermediate configuration in which all cubes are movable and a valid assignment can be computed. Then we show that the random cube motions of assignment resolution will assemble such a configuration with probability 1.

1. Starting in any nonterminal configuration, an intermediate configuration in which a valid assignment can be computed is reachable. For the purpose of this proof, let the intermediate configuration be a double line, which consists of two connected linear one-dimensional chains of cubes (such as the one shown in Fig. 19). Any configuration can be reconfigured into a double line according to Theorem 3, which shows completeness for homogeneous reconfiguration.[12] However, it applies here because a reconfiguration from any configuration to a double line can be treated as a homogeneous problem. This is true because for the purpose of assignment resolution, the absolute position of cubes in this double line is irrelevant. In a double line configuration, every cube is movable without disconnecting the configuration (see [61, 65]). Therefore, the movable set contains every cube $c_i \in \mathcal{C} \setminus \mathcal{C}_T$, i.e., every cube that is not at a target position. Because $\mathcal{R}$ is nonempty unless $\mathcal{C} = \mathcal{C}_T$ (see Lemma 3), at least one cube $m_i \in \mathcal{M}$ matches a position $p_i \in \mathcal{R}$ and a valid assignment can be found.

2. Using only random cubes motions, the probability of reaching a double line configuration approaches 1 as $t \to \infty$. That is because N cubes can only be arranged in a finite number of configurations assuming that at least one cube is fixed (see assumptions made in Section 2.3). Therefore, we can interpret a reconfiguration process as a finite state machine, in which each state corresponds to a configuration and each

[12] Note that a double line can always be assembled despite a subset of cubes being fixed to target positions due to permanence. This holds, because all occupied target positions form a connected sub-configuration $\mathcal{C}_o$ whose connectivity graph G_o is connected (as shown in Lemma 2 and 3).

transition to a cube motion. In such a state machine the probability of reaching a double line configuration from any state (i.e. configuration) is non-zero. Therefore, the random cube motions caused by assignment resolution will reconfigure the current configuration into a double line as $t \to \infty$, in which a valid assignment can be computed.

□

Note that in most practical cases a double line will not actually be assembled but assignment resolution will enable the computation of a valid assignment before. The reconfiguration process can then proceed towards assembling the target configuration. Referring to Fig. 19, the only reachable position in the first frame is a yellow position. Yet the only movable cube is red. Therefore, assignment resolution has to be applied to the red cube to move it to a random position. Once a yellow cube becomes movable the reconfiguration can proceed to assemble $\mathcal{C}_T$.

3.2.4 Completeness

Having defined the algorithm for heterogeneous reconfiguration and all its components in Section 3.1.3, Section 3.2.2, and Section 3.2.3, we can now prove its key property - completeness. In Theorem 1 that a successful homogeneous reconfiguration sequence cannot contain any holes. For heterogeneous systems, however, we cannot conclude that the absence of holes implies a successful reconfiguration because potential deadlocks can still prevent the completion of the reconfiguration sequence. Specifically, in this section we have shown that a deadlock can occur because of a lack of valid assignments. However, the assignment resolution algorithm introduced in Section 3.2.2 resolves these situations and guarantees the successful completion of heterogeneous reconfiguration.

Theorem 5. *Given any pair of heterogeneous, hole- and enclosure-free, two- or three-dimensional initial and target configurations $\mathcal{C}_I$ and $\mathcal{C}_T$, the heterogeneous reconfiguration algorithm (Algorithm 3) will compute a sequence of primitive cube motions that reconfigures $\mathcal{C}_I$ into $\mathcal{C}_T$ if such a sequence exists.*

Proof. As shown in Theorem 4, assignment resolution ensures that a valid assignment can be found as long as the reconfiguration has not been completed (because $\mathcal{M}$ and $\mathcal{R}$ are nonempty as long as $\mathcal{C} \neq \mathcal{C}_T$, as shown in Lemma 2 and Lemma 3). A valid assignment guarantees that progress is made towards assembling $\mathcal{C}_T$, i.e., the number of cubes occupying positions $p_i \in \mathcal{C}_T$ is strictly monotonically increasing in time. This property follows from Theorem 3, where we have shown for homogeneous systems that a valid assignment ensures that a path can be computed from $c_i \in \mathcal{M}$ to $p_i \in \mathcal{R}$. This statement holds because the planning space $\mathcal{N}(\mathcal{C})$ is connected and a complete path planner is used. Connectivity of $\mathcal{N}(\mathcal{C})$ depends on the avoidance of holes and enclosures which is guaranteed by Algorithm 2 and Theorem 2.

Given that a valid assignment can always be computed using assignment resolution and a valid assignment means progress towards assembling $\mathcal{C}_T$ is made, we can guarantee that $\mathcal{C}_T$ is indeed assembled. □

3.2.5 Results

This section presents numerical results based on reconfiguration sequences from random three-dimensional initial configurations to box configurations in Table 3. Additionally, a full reconfiguration sequence from a random configuration to a layered pyramid is shown in Fig. 20. In Table 3, the field *Size* refers to the size of the configurations, *Detected Holes* reports the number of holes and enclosures detected during reconfiguration, and *# of Resolutions* denotes the number of assignment resolution steps that occurred during each reconfiguration sequence. The results indicate a larger number of assignment resolution steps for small configurations. This can be attributed to the larger number of movable and reachable cubes in larger configurations, which makes it very likely that a valid assignment can be found. However, in this experiment it is most likely an artifact of the randomly generated initial configurations. The complete absence of holes and enclosures may also be an artifact of this specific set of randomly generated problem instances. However, the occurrence of holes and enclosures is very unlikely in general and requires multiple consecutive assignment or assignment resolution steps to arrange cubes in specific shapes. Nonetheless, holes and

Table 3: Reconfiguration planning results for reconfigurations from random to box configurations

Size	**Steps**	**Detected Holes**	**# of Resolutions**
10	33	0	3
20	69	0	1
30	107	0	0
40	150	0	0
50	233	0	1

enclosures are theoretically possible and need to be avoided in a complete heterogeneous self-reconfiguration algorithm as we have shown in this chapter.

3.3 Conclusions

This chapter has introduced basic graph theoretic notions required for self-reconfiguration planning, a system model including constraints and assumptions imposed on the system, as well as basic concepts of self-reconfiguration planning. Algorithms for centralized homogeneous as well as heterogeneous self-reconfigurable systems have been presented which have been shown to be provably complete. Despite being centralized approaches that use sequential planning and motion of cubes, the results in Section 3.1.6 and Section 3.2.5 show their feasibility for systems containing up to hundreds of cubes. Under mild constraints of enclosure and hole freedom Algorithm 1 and Algorithm 3 can be used for scenarios such as locomotion of self-reconfigurable systems, self-assembly, self-reconfiguration, and self-disassembly.

The next chapter makes use of the theory established for centralized reconfiguration in Section 3.1 and 3.2 to develop more scalable approaches through decentralization. Two approaches will be presented, one based on graph grammars and one that uses game theory to model the self-reconfiguration problem.

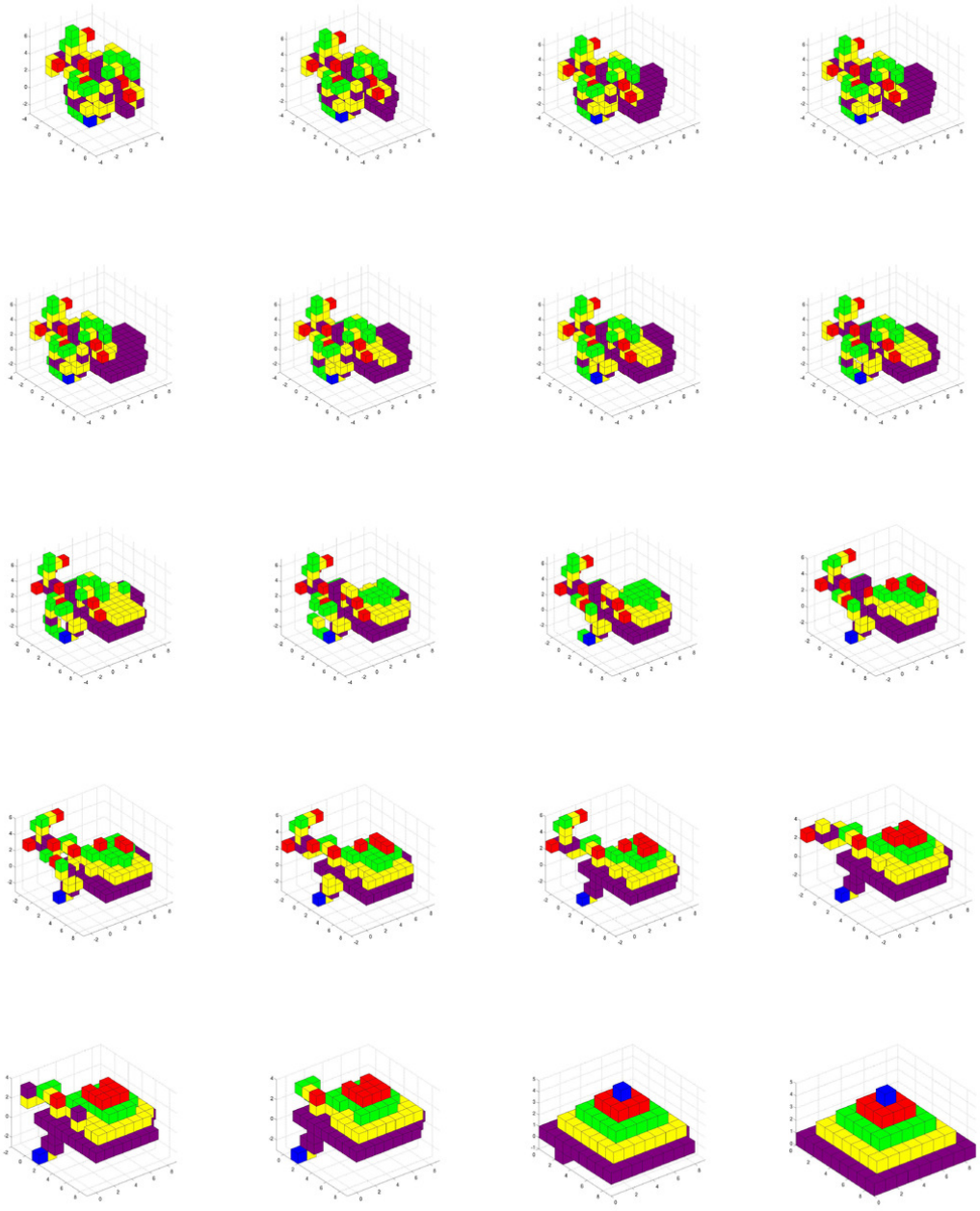


Figure 20: Example of a heterogeneous reconfiguration from a random three-dimensional configuration to a layered pyramid.

Chapter IV

DECENTRALIZED SELF-RECONFIGURATION PLANNING

This chapter outlines two decentralized approaches to homogeneous self-reconfiguration. Section 4.1 presents a method based on graph grammars which are automatically generated on a central node and then applied in a decentralized fashion. Section 4.2 formulates the self-reconfiguration problem as a potential game-based coverage problem and introduces a novel game-theoretic learning algorithm that solves it in a globally optimal fashion.[1] In this potential game, modules are represented as autonomous decision makers that can select actions based on local information. This chapter focuses specifically on homogeneous self-reconfiguration approaches, however, the presented methods can also be applied to heterogeneous systems with minor modifications. Graph grammars would have to be generated based on the set of paths computed through the algorithm in Section 3.2, while the utility functions in Section 4.2 would have to penalize a mismatch in agent positioning for heterogeneous agents.

A number of decentralized approaches to self-reconfiguration have been presented in the literature. Especially relevant to the results in this chapter are approaches such as the ones in [29, 30, 104], which employ cellular automata and manually designed local rules to model and reconfigure the respective systems. Similar work shown in [60] represents self-reconfiguration as a Markov decision process with state-action pairs that are continuously updated at runtime. Whereas these approaches are decentralized in nature, they are only able to accomplish locomotion of self-reconfigurable systems and do not generalize to the assembly of arbitrary configurations. A decentralized approach based presented in [66] automatically generates graph grammars that can be used to assemble arbitrary target configurations. Similar to Section 4.1, rulesets are precomputed for every target configuration.

[1] Note that optimality here is defined in terms of agents' utility value and not in terms of minimal cumulative distance traveled, as is usually the case.

However, the approach in [66] only allows target configurations that can be represented by acyclic connectivity graphs.

Approaches that most closely resemble the formulation in Section 4.2 are shown in [186, 192], in which stochastic self-assembly methods are applied to a robotic system. The shown online-assembly approach can robustly handle uncertainty and is able to pursue multiple paths in parallel. However, it depends on a central coordinating node.

4.1 A Graph Grammar-based Approach

Graph grammars are a popular graph rewriting mechanism that allow the manipulation and replacement of arbitrary subgraphs of a host graph. They were initially introduced in the 80s [56, 133] and found applications in image processing, software engineering and verification, layout algorithms, as well as image generation (see [57, 1]). More recently graph grammars have also been applied to distributed self-assembly on simulated systems [97, 66, 145] as well as modular robotic hardware implementations [14, 96, 94]. In [66, 145], graph grammars are automatically generate based on desired target configurations, while [97, 14, 96, 94]) relies on manually synthesized rule sets to form specific structures. Graph grammars as a tool for decentralization are an intuitive choice since the rules they are composed of are inherently local - checking for applicability of a rule and applying it only requires information from the local neighborhood in the graph. Graph grammars, however, are just one form of localized control of self-reconfiguration and a variety of rule-based reconfiguration approaches have been presented in the literature over the years.

Rule-based systems inspired by cellular automata, for example, were presented in [29, 28, 30]. The shown rulesets are designed manually and enable groups of modules to split and merge, climb over, or move around obstacles. Another rule-based control strategy was introduced in [26], where rules were automatically generated based on graph connectivity information. In [86], automatically generated rules were applied to two-dimensional self-assembly. Another approach to local control is shown in [59], where the reconfiguration problem is formulated as a Markov decision process and actions are mapped to lattice positions. Similarly, [194] formulates stochastic self-assembly as a Markov chain and designs

transition probabilities to optimize convergence rates.

In this section, we show how graph grammatical concepts can bridge the gap between global information that is available during planning and local information that is available during reconfiguration. Sets of paths obtained through centralized planning (see Section 3.1) are rewritten into rules that can be checked locally for applicability. Unlike rules based on connectivity information (such as [24, 26]), graph grammars offer fine-grained control over the applicability of rules and allow the encoding of additional information into the labels of the rules. The approach outlined in this section relies on some basic graph grammatical terminology (see [56, 133, 57, 95]) that will be briefly reviewed in the following section.

4.1.1 Graph Grammars

Graph rewriting systems like graph grammars require a host graph to operate on. In principle, graph grammars can be applied to unlabeled graphs, in which case they would have to rely exclusively on the graph structure. In this work however, we apply them to labeled graphs only, because labels allow the encoding of additional information and thus the opportunity of applying graph grammars in a more fine-grained and targeted fashion. A thorough review of graph grammatical concepts with applications to modular robotics is given in [95].

Definition 12. *Let Σ be an alphabet. Then the tuple $G = (V, E, l_V, l_E)$ is called a labeled graph composed of N nodes with a vertex set $V = \{v_1, v_2, \ldots, v_N\}$, an edge set $E = V \times V$, and two labeling functions $l_V : V \to \Sigma$ and $l_E : E \to \Sigma$. A label is an element from the alphabet Σ and assigned to all (or a subset of all) vertices and edges.*[2]

We will use labeled graphs extensively in the definition, generation, and application of graph grammatical rules, which are the basic components of graph grammars.

Definition 13. *A graph grammatical rule $r = \{g_l, g_r\}$ consists of two labeled graphs: a left-hand side g_l and a right-hand side g_r. It describes a transformation of a graph G_S, that is isomorphic to g_l, from G_S to g_r. Note that $V_{g_l} = V_{g_r}$ (i.e. the vertex set of left and right hand side are the same) and the size of the rule is given by $|V_{g_l}| = |V_{g_r}|$.*

[2] *Note that this definition of labeled graph differs from [95], which defines labels only for vertices.*

Note that typically G_S is a subgraph of the full host graph. Therefore graph grammatical rules operate on a local subset of vertices. The diameter of g_l and g_r[3] is a measure of how local a rule is - the smaller the diameter the more local the rule. Note that this notion of locality differs from [95], where locality is defined as the number of nodes in the vertex set V_{g_l} and V_{g_r} of the left and right hand side of a rule. In other words, the rule size determines the total number of nodes a module has to communicate to execute a rule. Our notion of locality is based on the number of hops required to check a rule for applicability (since the diameter measures a distance in hops). For both notions of locality, however, larger rules require more communication between agents because they need to determine whether their neighborhood structure matches the left hand side of a rule. A set of rules that operate on a graph G is furthermore called a graph grammar Φ. A rule $r \in \Phi$ can be applied to G only when it is applicable.

Definition 14. *A rule $r \in \Phi$ is applicable to G if there exists a subgraph G_S of G that is isomorphic to g_l. This is also denoted as $G_S \cong g_l$.*

If a rule r is applicable to G, its application yields a new graph $G_i \xrightarrow{r} G_{i+1}$, where G_{i+1} results from G_i by replacing the subgraph G_S with g_r. Each step in the reconfiguration process yields a graph G_i that is part of a trajectory, i.e. a finite or infinite sequence of graphs $\{G_i\}_{i=0}^{k}$ s.t. there exists a sequence of applicable rules $\{r_i\}_{i=0}^{k-1}$ where $r_i \in \Phi$ and $G_i \xrightarrow{r_i} G_{i+1}$. Each graph G_i as part of a trajectory is called reachable by the system (G_0, Φ). A reachable graph can be temporary, such that some rule $r \in \Phi$ is applicable to it, or stable (see [95]).

Definition 15. *A graph G is stable, if no rule in Φ is applicable to it.*[4]

Stable graphs play an important role in this section. Theorem 6 proves that the graph representing $\mathcal{C}_T$ is the only reachable stable graph given a system (G_0, Φ) where G_0 is the graph representing the initial configuration $\mathcal{C}_I$ and Φ is an automatically generated graph grammar.

[3] The diameter of a graph G is the longest shortest path between any pair of vertices $v_i, v_j \in G$.

[4] *Note, however, that this does not mean that no more rules are applicable to G, merely that it is left unchanged by the application of any further rules.*

4.1.2 Problem Setup

This graph grammar-based approach rests on the assumption that a set of individual module paths have already been computed and are available as input to the graph grammar generator. Specifically, this section uses paths that were generated using the homogeneous self-reconfiguration method shown in Section 3.1 but can be equivalently used with paths generated for heterogeneous systems (as shown in Section 3.2). Given a set of module paths as input, this section focuses on the automated generation of rulesets that encode this set of paths as graph grammatical rules (see Section 4.1.4). Therefore, the decentralized execution of the generated graph grammar yields the exact same motion sequence as the centralized method in Section 3.1.2.

Graph grammar generation relies on the representation of the current state of the self-reconfigurable system as a labeled graph (as will be shown in Section 4.1.3). The execution of the ruleset however, uses the original reconfigurable system represented by the sliding cube abstraction from Section 2.3. More specifically, each cube will be treated as an autonomous agent that is responsible for periodically checking the ruleset for currently applicable rules and then executing them. The rule applicability check can be done with local information where local is defined as a distance of at most two hops (with hops used in the same sense as in Def. 5). Each rule therefore describes a small neighborhood of the current cube and as such a rule can only manipulate cubes in that neighborhood (either through position or label updates). The rule execution is done in a decentralized fashion during which each cube can only access neighborhood information and the ruleset (see Section 4.1.5). In addition to the assumptions and constraints stated for centralized homogeneous reconfiguration in Section 3.1, we clarify another set of assumptions that govern the rule generation and execution in this section.

Assumptions

- A set of module paths that reconfigures an initial configuration $\mathcal{C}_I$ into a target configuration $\mathcal{C}_T$ is available as input.

- Each rule can encode a cube motion, label updates, or both.
- Each rule can move a single cube at a time but is able to update the labels of all or a subset of the cubes in the local neighborhood.
- Since valid paths are encoded in the ruleset, every rule obeys the motion model as well as any other constraints imposed in Section 2.3, Section 3.1.1 and Chapter 3 in general.

4.1.3 Rule Structure

The two labeled graphs g_l and g_r of a rule in our system are derived from local neighborhoods of cubes. Therefore, a mapping from connected sets of cubes to labeled graphs is required.

Definition 16. *The mapping $f : \mathbb{Z}^{dN} \rightarrow \mathcal{G}_N^L$ (with $d \in \{2,3\}$) maps from the space of configurations composed of N cubes to the space of labeled connectivity graphs on N nodes such that $f(\mathcal{C}) = G(V, E, l_G)$. The vertex and edge set are computed according to Def. 1 (unlabeled connectivity graphs) while the edge and vertex labels are defined as follows.*

$$
\begin{aligned}
l_G(v_i) &= l(c_i) && \text{with } v_i \in V, c_i \in \mathcal{C} \\
l_G(e_{ij}) &= \begin{cases} b_k & \text{if } (c_i - c_j) \cdot b_k < 0 \\ -b_k & \text{o.w.} \end{cases} && \text{with } e_{ij} \in E
\end{aligned}
$$

Here, b_k is a basis vector of the lattice $\mathbb{Z}^d$.

This definition implies that vertex labels are equivalent to the labels of the cubes, while edge labels preserve information about the two- or three-dimensional structure of the configuration. Therefore, the labels of edges e_{ij} and e_{ji} differ in sign. The mapping f preserves the entire state of a configuration in the labeled graph. As such, the inverse $f^{-}1 : \mathcal{G}_N^L \rightarrow \mathbb{Z}^{dN}$ of a labeled connectivity graph can be computed and a two- or three-dimensional configuration can be recovered. Note that edge labels are computed automatically based on the relative position of the cubes to each other. Edge labels are not used in the a rule's label update mechanism but need to be part of the labeled graph such that the mapping f is invertible (through preserving the full two- or three-dimensional structure of the configuration in the labeled graph).

For the definition of the left- and right-hand side g_l and g_r consider a path $p_i = \{p^0, p^1, \ldots, p^m\}$ of cube c_i (with $c_i = p^0$ and $m = |p_i|$) and the following neighborhood.

Definition 17. *Let the neighborhood*

$$\mathcal{N}_2(c_i) = \{c_j : \|c_i - c_j\|_{L_2} < 2, c_i, c_j \in \mathcal{C}\} \tag{10}$$

denote the motion neighborhood of cube c_i, i.e. the set of all cubes that are within one primitive motion of c_i.[5]

The elements p_i^j of path p_i (with $j \in \{0, \ldots, |p_i|\}$) represent the positions of cube c_i as it moves along path p_i. Given these prerequisites, the components g_l and g_r of a rule r can now be defined as follows (note that here $j \in \{0, \ldots, |p_i| - 1\}$).

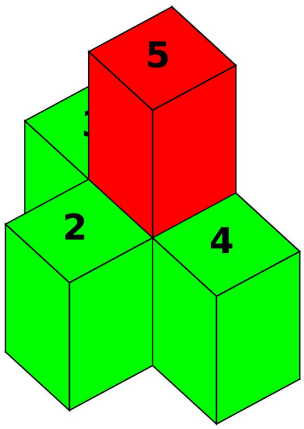
$$g_l = f\left(\mathcal{N}_2(p_i^j)\right) \tag{11a}$$

$$g_r = f\left(\mathcal{N}_2(p_i^{j+1})\right) \tag{11b}$$

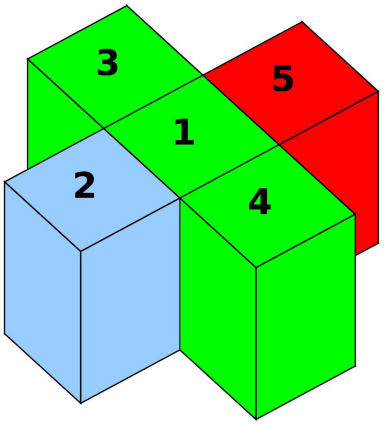
The application of a rule r to a subgraph G_S yields a new graph G'_S. The changes in the edge and label set described by $G = (V, E, l_G) \xrightarrow{r} (V, E', l'_G) = G'$ represent a motion in the configuration space, i.e. cube c_i has moved from position p_i^j to position p_i^{j+1}.

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 (an example is shown in Listing 4.1 and the corresponding three-dimensional and graph representation in Fig. 21). Since the labels of g_l and g_r in the generated rules are essential in guaranteeing the properties of our reconfiguration approach, we will present a detailed description of their structure. Listing 4.1 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 4.1). 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

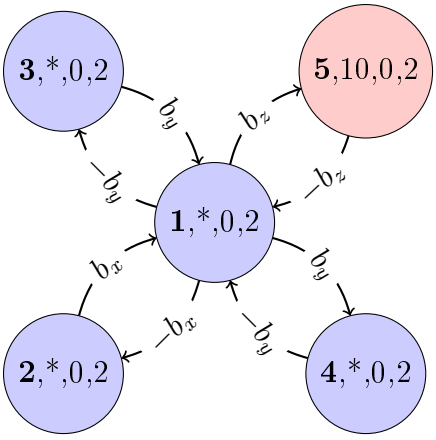
[5] *This definition differs slightly from Def. 5 in that the L_2 norm is used here instead of the L_1 norm.*



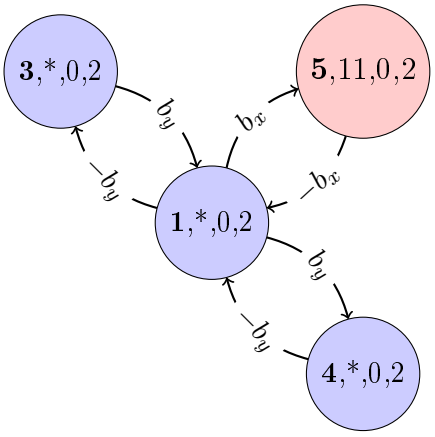
(a) 3D representation of the left hand side



(b) 3D representation of the right hand side



(c) Graph representation of the left hand side of the rule



(d) Graph representation of the right hand side of the rule

Figure 21: Examples of motion rule showing the three-dimensional representation of the rule in the top row and the labeled graph representation in the bottom row. The active cube is shown in red (with ID 5), its motion neighbors according to Def. 17 in green and cubes outside its motion neighborhood in light blue. A nodes ID is shown in bold font as the first component of each node label.

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 $G_0 = f(\mathcal{C}_I)$ and the label update mechanism through rules are designed such that only one motion rule (see Section 4.1.4) is applicable to the current configuration $\mathcal{C}$ at any given time. Therefore, the reconfiguration sequence is unambiguous and deterministic.

```
ID                  : 10
gl_struct           : [1x1 struct]
gl_labels           : '5,10,0,2'
gr_struct           : [1x1 struct]
gr_labels           : '5,11,0,2'
update neighbors    : []
```

Listing 4.1: Example of a motion rule. The application of the shown rule with ID 10 changes the edge set of a local neighborhood of node 5 (the edge set is specified by *gl_struct* and *gr_struct*) and updates the labels of node 5 such that the next applicable rule is the rule with rule ID 11 (see field *gl_labels and gr_labels*).

4.1.4 Rule Generation

This section outlines the automatic generation of a graph grammar Φ that unambiguously reconfigures the initial into the target configuration by repeatedly rewriting the host graph $G = f(\mathcal{C})$. The result is a trajectory given by $f(\mathcal{C}_I) \xrightarrow{\Phi} f(\mathcal{C}_T)$. Rules are generated based on a set of cube paths $\mathcal{P}$ that can be computed using the approaches shown in Chapter 3. For each path $p_i \in \mathcal{P}$ a sub-ruleset $R_{p_i} = \{\{r_{m,j}\}_{j=1}^{|p_i|}, r_{pr}, r_f\}$ is generated that consists of $|p_i|$ motion rules $r_{m,j}$ (where $|p_i|$ is the path length), one flooding activation rule r_f, and one propagation rule r_{pr}. The graph grammar Φ is then the set of all sub-rulesets $\Phi = \left\{\{R_{p_i}\}_{i=1}^{|\mathcal{P}|}\right\}$. The labels of each rule (i.e. the labels of g_l and g_r) contain strictly monotonically increasing rule IDs which result in a sequential application of rules.

Additionally each rule's labels contain the ID of the cube it applies to ensuring that only one cube in the configuration can apply each rule (for motion and flooding activation rules). Propagation rules are wildcard rules that can be applied by every cube in the configuration.

Rule Types Three types of rules are generated - motion, flooding activation, and propagation rules (introduced in [145]). Intuitively, a motion rule represents a primitive cube motion by rewriting the edge set and labels of a local neighborhood of the host graph through its application.[6] A flooding activation rule triggers at the end of each path, changes the labels of the applying cube, and starts the propagation process. Propagation rules inform the whole configuration about the conclusion of the latest path, which is required because paths are strictly ordered and the start of path p_{i+1} depends on the conclusion of path p_i. Note that flooding activation and propagation rules do not result in cube motions but only in label updates.

As the name suggests, a motion rule results in the motion of a cube c_i and therefore modifies the edge set of its neighborhood. In addition a cube c_i's labels are updated such that the next rule is applicable to it and the flooding flag is reset.

Definition 18. *A motion rule r_m moves a cube c_i by rewriting the host graph $G = (V, E, l_G)$ according to $(V, E, l_G(v_i)) \xrightarrow{r_m} (V, E', l'_G(v_i))$ with edge and label updates computed as follows.*

$$E' = (E - E_{g_l}) \cup E_{g_r}$$

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

$$\begin{aligned} l'_G(v_i) \rightarrow \textit{rule ID} &= l_G(v_i) \rightarrow \textit{rule ID} + 1 \\ l'_G(v_i) \rightarrow \textit{flooding} &= 0 \end{aligned}$$

For each pair of consecutive positions p_i^j and p_i^{j+1} on path p_i, the neighborhood structure of the active cube is computed (according to Eqn. 11) and stored in the labeled graphs g_l and g_r of the corresponding motion rule. Once a cube c_i completes its path, a flooding

[6] Note that according to Definition 1 an edge between nodes in the connectivity graph exists only for cubes located at neighboring grid positions. Therefore a change to the edge set requires the motion of a cube.

activation rule triggers an update process that uses a propagation rule to inform the whole configuration of the conclusion of path p_i of cube c_i. A flooding activation rule updates the labels of the current cube only. Therefore, no motion occurs and neighboring cubes are not affected by the label update.

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 0 to 1, which activates the corresponding propagation rule. More formally, r_f rewrites the host graph $G = (V, E, l_G)$ according to $(V, E, l_G(v_i)) \xrightarrow{r_f} (V, E, l'_G(v_i))$ with the edge set remaining unchanged and the labels updated from $l_G(v_i)$ to $l'_G(v_i)$ as follows:*

$$l'_G(v_i) \rightarrow \textit{rule ID} = l_G(v_i) \rightarrow \textit{rule ID} + 1$$

$$l'_G(v_i) \rightarrow \textit{flooding} = 1$$

Once a flooding rule is triggered, a propagation rule ensure that the whole configuration is informed of the latest finished path. Similar to flooding activation rules, a propagation rule only updates labels and does not change the edge set of G. Therefore, no motion occurs, only the labels of the current node itself as well as all of its neighbors are updated. A propagation rule is a wildcard rule, i.e. it applies to every node irrespective of its node ID if all other label fields agree. Therefore, one rule is sufficient to update the entire configuration after a single path has concluded.

Definition 20. *A propagation rule r_p updates the current node v_i's labels by setting its flooding flag from 1 to 0 and incrementing its last path field. It also sets the flooding flag of all its neighbors $v_j \in f(\mathcal{N}_2(c_i, \mathcal{C}))$ to 1 so that the same rule r_p is applicable to them. More formally, r_p rewrites the host graph $G = (V, E, l_G)$ according to $(V, E, l_G(v_i, v_j)) \xrightarrow{r_p} (V, E, l'_G(v_i, v_j))$ with the labels changing as follows.*

$$l'_G(v_i) \rightarrow \textit{rule ID} = l_G(v_i) \rightarrow \textit{rule ID} + 1$$

$$l'_G(v_i) \rightarrow \textit{path} = l_G(v_i) \rightarrow \textit{path} + 1$$

$$l'_G(v_i) \rightarrow \textit{flooding} = 0$$

For all adjacent nodes $v_j \in f(\mathcal{N}_2(c_i, \mathcal{C}))$ the following label updates are performed.

$$l'_G(v_j) \rightarrow \textit{flooding} = 1$$

To avoid propagation messages being passed back and forth between nodes, only those rules with a higher rule ID than the one currently stored in a node's labels are executed. This avoids the repeated execution of the same propagation rule by a single node because the executing node's rule ID field is incremented to a value larger than the propagation rule's ID. Below, we summarize how the labels of g_l and g_r of each rule are computed. In general, for a given position p_i^j on path p_i, the labels of g_l are defined as follows:

$$l_G(g_{l_{i,j}}) = \begin{cases} l_G(g_{r_{i,j-1}}) & \text{if } j > 1 \\ l_G(g_{r_{i-1,|p_{i-1}|}}) & \text{if } j = 1, i > 1 \\ l_{G,init} & \text{otherwise} \end{cases} \tag{12}$$

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.

$$l_G(g_{r_{i,j}}) = \begin{cases} l_G(g_{l_{i,j}}) \xrightarrow{r_m} l_G(g_{r_{i,j}}) & \text{for motion rules} \\ l_G(g_{l_{i,j}}) \xrightarrow{r_f} l_G(g_{r_{i,j}}) & \text{for flooding rules} \\ l_G(g_{l_{i,j}}) \xrightarrow{r_p} l_G(g_{r_{i,j}}) & \text{for propagation rules} \end{cases} \tag{13}$$

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. For every path p_i, $|p_i|$ motion rules, one flooding activation rule, and one propagation rule are generated, resulting in a ruleset $R_{p_i} = \{\{r_{m_i}\}_{i=1}^{|p_i|}, r_p, r_f\}$. This rule generation process is repeated for every path p_i, $i \in \{1..|P|\}$) until the reconfiguration sequence is completed, i.e. until the target configuration $\mathcal{C}_T$ has been assembled.

4.1.5 Ruleset Execution

Given a system (G_0, Φ), the application of Φ results in the reconfiguration sequence $G_0 \xrightarrow{r_1} G_1 \xrightarrow{r_2} G_2 \xrightarrow{r_3} \ldots \xrightarrow{r_N} G_{stable}$, where $G_0 = f(\mathcal{C}_I)$, $G_{\text{stable}} = f(\mathcal{C}_T)$ (the fact that the only stable graph is $G_{\text{stable}} = f(\mathcal{C}_T)$ will be shown in Section 4.1.6). To accomplish this reconfiguration, every node $v_i \in G$ periodically checks the ruleset for applicable rules. If a cube's local neighborhood $G_S = f(\mathcal{N}_2(c_i))$ is isomorphic to the left-hand side g_l of some

rule $r \in \Phi$, r is applied to the current node $v_i = f(c_i)$. The application of r rewrites the subgraph G_S into g_r. In the case of motion rules, such rewriting results in a motion in the configuration space. The execution of the last motion rule r_m of a path p_i triggers the application of a flooding activation rule r_f which in turn triggers a propagation rule r_p. The repeated application of r_p to every node $v_i \in V$ updates every node's local state about the completion of the latest path.[7]

This process is repeated until every path p_i is completed and no more rules in Φ are applicable to any node $v_i \in V$, at which point a stable graph is reached. Unlike the centralized path planning and ruleset generation stage, decentralized execution relies on local neighborhood information only. Therefore, rulesets can be executed in a highly parallel fashion with each module checking simultaneously for applicable rules.

4.1.6 Convergence

The sequential nature of the generated rules ensures that the only reachable, stable graph as defined in Def. 15 is the graph representing the desired target configuration $f(\mathcal{C}_T)$. In other words, the ruleset Φ results in a uniquely determined sequence of cube motions that unambiguously assembles $\mathcal{C}_T$ (see [145]). The following theorem formalizes this statement and proves convergence to $\mathcal{C}_T$.

Theorem 6. *The graph $G_T = f(\mathcal{C}_T)$ is the only reachable, stable graph of the ruleset Φ when applied to the initial graph $G_0 = f(\mathcal{C}_I)$.*

Proof. By assumption, the graph grammar generator is given a set of paths that is guaranteed to assemble the target configuration $\mathcal{C}_T$ from a given initial configuration $\mathcal{C}_I$ (by Theorem 3). What remains to be shown is that the execution of the generated graph grammar obeys the same sequence of module motions and paths as the centralized planner in Section 3.1. To show this result, we will rely on the strictly monotonically increasing and globally unique rule numbers.

Rules are generated for each path p_i in sequence, i.e. $|p_i|$ motion rules, one flooding activation rule, and one activation rule are generated before path p_{i+1} is used. Therefore,

[7] Note that propagation rules are wildcard rules and hence applicable to every node in the graph.

each rule $r_{i,j} \in R_{p_i}$ (where $j \in \{1, \ldots, |R_{p_i}|\}$ and $R_{p_i} = \{\{r_{m,k}\}_{k=1}^{|p_i|}, r_p, r_f\}$ is the sub-ruleset of path p_i) has a lower rule number than any rule $r_{i+1,k} \in R_{p_{i+1}}$ (where $k \in \{1, \ldots, |R_{p_{i+1}}|\}$). These rule numbers are unique, strictly monotonically increasing, and are encoded in the labels of $r_{i,j}$. As a result, the applicability of rule $r_{i,j}$ depends on the successful execution of rule $r_{i,j-1}$ (for any rule but the first rule $r_{i,1}$ in each sub-ruleset $R_{p_{i+1}}$) and on $r_{i-1,|R_{p_{i-1}}|}$ (for the first rule of each path p_i). As such, the execution of motion rules clearly cannot change the sequence of motions. Flooding activation rules are executed only once at the end of each path by the currently active node and have no influence on other modules in the system. Propagation rules also execute a single time, but on every module in the system. Therefore propagation rules ensure that after N executions (once on each module) the entire configuration is informed as to which path has just been completed (through the *last path* field) and which path has to be executed next.

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 $(V, E, l_G) = f(\mathcal{C}^{\mathcal{T}})$. □

4.1.7 Results

This section presents rule generation results based on paths generated in Section 3.1 (homogeneous reconfiguration planning). As in Section 3.1, the reconfiguration sequences and rule sets were generated through simulation in Matlab. Configurations ranged in size from 100 to 500 modules and were either randomly generated or arranged in box configurations (examples are shown in the insets of Fig. 16 and Fig. 17). The parameters and results of these simulations are summarized in Table 4 and Table 5. In these tables, the field *Size* refers to the number of modules in the configuration, *Overlap* is the number of initially overlapping modules, *Path Length* is the total number of motions of all modules to achieve the desired reconfiguration, and *Rules* is the total number of rules in the ruleset.

As shown in Table 4 and Table 5 the size of the ruleset increases approximately linearly with the number of modules and shows a similar trend as the cumulative path length. Given

Table 4: Reconfiguration planning and rule generation results for overlapping box configurations

Size	Overlap [N]/[%]	Path Length	Rules
100	30 / 30%	837	977
200	60 / 30%	1543	1823
300	90 / 30%	2426	2846
400	120 / 30%	3279	3839
500	150 / 30%	4275	4975

Table 5: Reconfiguration planning and rule generation results for overlapping random configurations

Size	Overlap [N]/[%]	Path Length	Rules
100	36 / 36%	352	480
200	114 / 57%	403	575
300	157 / 52.3%	893	1179
400	182 / 45.5%	1674	2106
500	231 / 46.2%	2327	2865

the rule generation approach the rule set size can be computed as follows.

$$|\Phi| = \text{cumulative path length} + 2(|\mathcal{C}_I| - |\mathcal{O}|)$$

Essentially, the ruleset size is determined by the cumulative path length while an additional two rules are generated for every completed path.

4.2 Game-theoretic Reconfiguration

In this section, we present a scalable and fully distributed approach to homogeneous self-reconfiguration that does not rely on a central decision maker, requires no precomputation, and uses only limited communication between agents. The homogeneous self-reconfiguration problem is formulated as a constrained potential game and solved in a provably globally optimal fashion using a novel game-theoretic learning algorithm.[8] This learning algorithm induces a Markov process that is guaranteed to converge to the unique stochastically stable

[8]Note that optimality here is defined in terms of agents' utility value and not in terms of minimal cumulative distance traveled, as is usually the case.

state that maximizes the global potential, i.e. the desired target configuration. Convergence to the target configuration is provably achieved even though each agent acts as a purely self-interested individual decision maker with local information only. Furthermore, decision making requires no (in the two-dimensional case) or limited communication (in the three-dimensional case). The learning algorithm relies on a parameter τ that is commonly referred to as the learning rate and determines the tradeoff between exploration of the state space and exploitation (i.e. maximization of an agent's utility). This section introduces a learning strategy that uses a fixed learning rate τ, which will be compared to an adaptive learning rate in Chapter 5. The problem setup can be briefly summarized as follows and will be rigorously defined in Section 4.2.1.

- The environment $\mathcal{E}$ is a finite two- or three-dimensional discrete grid, i.e. $\mathcal{E} \subset \mathbb{Z}^2$ or $\mathcal{E} \subset \mathbb{Z}^3$.
- N agents $P = \{1, 2, \ldots, N\}$ move in discrete steps through that grid.
- An agent's full action set A_i contains every grid position $a_i \in \mathcal{E}$.
- Each agent has a restricted action set $\mathcal{R}_i$ which contains only a subset of all its possible actions A_i.
- An agent's utility or reward $U_i(a \in A)$ is inversely proportional to the distance to the target configuration.
- The configuration $\mathcal{C}$ composed of all N agents does not have to remain connected.

4.2.1 Problem Formulation

The work presented in this section relies on the sliding cube abstraction shown in Section 2.3, in which individual agents are represented as cubic modules that move through a discrete lattice or environment $\mathcal{E} = \mathbb{Z}^d$ in discrete steps (with d being the dimensionality of the environment $d \in \{2, 3\}$). The motion model is equivalent to Section 2.3, however, an agent does not require a substrate of other agents to execute a motion. An agent i's current state or action a_i (in a game-theoretic sense) is a position in the lattice $a_i \in \mathbb{Z}^d$. Note that

an agent's action is equivalent to its position in the lattice. Cubes can be thought of as geometric embeddings of the agents in our system. Note that in this section, we model a homogeneous self-reconfigurable system. Hence, all agents have the same properties and are completely interchangeable.

The homogeneous self-reconfiguration is furthermore formulated as a potential game (see [128]), which is a game structure amenable to globally optimal solutions and can be decentralized in a straightforward fashion. Generally, a game is specified by a set of players $i \in P = \{1, 2, \ldots, N\}$, a set of actions $\mathcal{A}_i$ for each player, and a utility function $U_i(a) = U_i(a_i, a_{-i})$ for every player i. In this notation, $a = (a_1, a_2, \ldots, a_N)$ denotes a joint action profile of all N players, and a_{-i} is used to denote the actions of all players other than agent i. In a constrained game, the actions of agents are constrained through their own and other agents' actions. In other words, given an action set $\mathcal{A}_i$ for agent i, only a subset $\mathcal{R}_i(a)) \subset \mathcal{A}_i$ is available to agent i. A constrained potential game is furthermore defined as follows.

Definition 21. *A constrained exact potential game (see [201]) is a tuple* $G = (\mathcal{P}, \mathcal{A}, \{U_i(.)\}_{i\in\mathcal{P}}, \{R_i(.)\}_{i\in\mathcal{P}}, \Phi(A))$, *where*

- $\mathcal{P} = \{1, \ldots, N\}$ *is the set of* N *players*
- $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_N$ *is the product set of all agents' action sets* $\mathcal{A}_i$
- $U_i : A \to \mathbb{R}$ *are the agents' individual utility functions*
- $R_i : \mathcal{A} \to 2^{\mathcal{A}_i}$ *is a function that maps a joint action to agent* i*'s restricted action set*

Additionally, the agents' utility functions are aligned with a global objective function or potential $\Phi : \mathcal{A} \to \mathbb{R}$ *if for all agents* $i \in \mathcal{P}$, *all actions* $a_i, a_i' \in R_i(a)$, *and actions of other agents* $a_{-i} \in \prod_{j\neq i} \mathcal{A}_j$ *the following is true*

$$U_i(a_i', a_{-i}) - U_i(a_i, a_{-i}) = \Phi(a_i', a_{-i}) - \Phi(a_i, a_{-i})$$

The last condition of Def. 21 implies an alignment of agents' individual incentives with the global goal. Therefore, under unilateral change (only agent i changes its action from a_i to a_i') the change in utility for agent i is equal to the change in the global potential Φ. This is

a highly desirable property since the maximization of all agents' individual utilities yields a maximum global potential. Additionally, for potential games (see [117, 118, 128]), it can be guaranteed that there exists at least one pure Nash equilibrium, which is the action profile maximizing Φ. We can now formulate the self-reconfiguration problem in game-theoretic terms and show that it is indeed a constrained potential game.

Definition 22. *Game-theoretic self-reconfiguration can be formulated as a constrained potential game, where the individual components are defined as follows:*

- *The set of players* $\mathcal{P} = \{1, 2, \ldots, N\}$ *is the set of all* N *agents in the configuration.*
- *The action set of each agent* $\mathcal{A}_i = \mathcal{E} \subset \mathbb{Z}^d$ *is a finite set of discrete lattice positions (i.e. every position in the environment* $\mathcal{E}$*)*
- *The restricted action sets* $R_i(a \in A)$ *are computed according to Section 4.2.2.*
- *The utility function of each agent is* $U_i(a_i) = \frac{1}{d(a_i)+1}$*. Here,* $d(a_i)$ *specifies the minimum distance to the target configuration* $\mathcal{C}_T$ *as* $d(a_i) = \min_{a_j \in \mathcal{C}_T} \|a_i - a_j\|_{L_2}$*.*
- *The global potential* $\Phi(a \in A) = \sum_{i \in \mathcal{P}} U_i(a_i)$*.*

Note that the utility of an agent is independent of all other agents' actions and depends exclusively on its distance to the target configuration. It is this decoupled expression of agents' utility functions that allows the expression of the global potential as a sum of individual utilities. The coupling between agents is then introduced through an agent's restricted action set computation (see Section 4.2.2), which is constrained by its own as well as other agents' actions. The goal of the game-theoretic self-reconfiguration problem is to maximize the potential function, i.e.

$$\max_{a \in \mathcal{A}} \Phi(a) = \max_{a \in \mathcal{A}} \sum_{i \in \mathcal{P}} U_i(a_i) \tag{14}$$

This can be interpreted as a coverage problem where the goal of all agents is to cover all positions in the target configuration. Therefore maximizing the potential is equivalent to maximizing the number of agents that cover target positions $a_i \in \mathcal{C}_T$. The following propositions shows that this formulation indeed yields a potential game.

Proposition 1. *The self-reconfiguration problem in Def. 22 constitutes a constrained potential game with* $\Phi(a) = \sum_{i \in \mathcal{P}} U_i(a)$ *and* $U_i(a) = \frac{1}{d(a_i)+1}$.

Proof. Note that the dependence of an agent's utility exclusively on its own state allows us to express agent i's utility as $U(a_i, a_{-i}) = U(a_i)$ and the global potential $\Phi(a_i, a_{-i}) = U(a_i) + \sum_{j \in \mathcal{P} \setminus \{i\}} U_j(a_j)$. Therefore the desired result follows directly from the defining equation of potential games.

$$U_i(a_i') - U_i(a_i) = \Phi(a_i', a_{-i}) - \Phi(a_i, a_{-i})$$

□

As we will see in Section 4.2.5, this potential game structure allows us to derive a decentralized version of the presented learning algorithm.

4.2.2 Action Set Computation

A core component of constrained potential games is the computation of restricted action sets. Unlike in previous work (for example [118, 201]), agents in our setup are constrained not just by their own actions, but also those of others. In this section we present methods for computing restricted action sets that obey motion constraints as well as collision constraints imposed by other agents.

2D reconfiguration In the two-dimensional case agents are restricted to motions on the xy-plane. Unlike in previous work (see [145, 146]) where we required a configuration to remain connected at all times, in this work, agents are allowed to disconnect from all (or a subset of) other agents. This approach enables agents to separate from and merge with other agents at a later time. To formalize this idea, we first review some graph theoretic concepts. Note that the following definitions have been adapted from similar concepts introduced in Section 3 to fit the game-theoretic setup in this section.

Definition 23. *Let* $G = (V, E)$ *be the graph composed of* N *nodes with* $V = \{v_1, v_2, \ldots, v_N\}$, *where node* v_i *represents agent* i*'s position* a_i. *Then* G *is called the connectivity graph of*

configuration $\mathcal{C}$ *if* $E = V \times V$ *with* $e_{ij} \in E$ *if* $\|a_i - a_j\| = 1$ *(where* $a_i, a_j \in \mathcal{C}$ *are the positions of agents* $i, j \in \mathcal{P}$*).*

This definition implies that two nodes v_i, v_j in the connectivity graph are adjacent, if agent i and j are located in neighboring grid cells. Note that a connectivity graph can be computed for any set of grid positions, whether these positions are occupied by agents or not. We furthermore use the notions of paths on graphs and graph connectivity in the usual graph theoretic sense. Note that G is not necessarily connected since (groups of) agents can split off. Based on the connectivity graph G and the current joint action a, we now define the function $R_i : \mathcal{A} \to 2^{\mathcal{A}_i}$, which maps from the full joint action set to a restricted action set for agent i and is based on the following two definitions of primitive actions sets.

Definition 24. *The set of all currently possible sliding motions* $\mathcal{M}_s$ *is defined as follows (where* $m_s = a_i' - a_i$*).*

$$\mathcal{M}_s = \left\{ a_i' \in \mathbb{Z}^d \setminus a_{-i} : \|m_s\|_{L_1} = 1 \right\} \tag{15}$$

Definition 25. *The set of all currently possible corner motions* $\mathcal{M}_c$ *is defined as follows (where* $j \in [1, \ldots, d]$ *and* $m_c = a_i' - a_i$*).*

$$\mathcal{M}_c = \left\{ a_i' \in \mathbb{Z}^d \setminus a_{-i} : \|m_c\|_{L_1} = 2 \ \wedge \ m_{c,j} \in \{0, 1\} \right\} \tag{16}$$

Note that $\mathcal{M}_s$ and $\mathcal{M}_c$ in Def. 24 and Def. 25 are equally applicable to 2D and 3D. These definitions encode the motion model and collision avoidance constraint outlined in Section 2.3. They allow us to define the restricted action set in two dimensions as follows.

Definition 26. *The two-dimensional restricted action set is given by* $R_i^{2D}(a) = \mathcal{M}_s \cup \mathcal{M}_c$*.*

This definition ensures that agent i can only move to unoccupied neighboring grid positions a_i' through sliding or corner motions (or stay at its current position a_i). All other agents replay their current actions a_{-i}.

3D reconfiguration Whereas in the 2D case agents were allowed to move to all unoccupied neighboring grid cells regardless of connectivity constraints, in the three-dimensional

case we introduce the requirement of *groundedness*. Intuitively, groundedness requires agents to remain connected to a ground plane and prevents agents from occupying arbitrary positions in the three-dimensional lattice. As such, groundedness enforces a certain level of cohesion among agents while at the same time allowing them to disconnect from the main configuration and merge at a later time. In this sense, groundedness affords agents more flexibility in selecting their actions than global connectivity, which requires all agents to remain connected at all times. An agent is furthermore immobile, if executing an action would remove its own groundedness or that of any of its neighbors. The required notion of a ground plane is defined as follows.

Definition 27 (Ground Plane)**.** *The ground plane is the set* $S_{GP} = \{s \in \mathcal{E} : s_z = 0\}$ *where* $\mathcal{E}$ *is a finite environment* $\mathcal{E} \subset \mathbb{Z}^3$ *and the corresponding connectivity graph is* $G_{GP} = (V_{GP}, E_{GP})$ *with* $e_{ij} \in E_{GP}$ *if* $\|s_i - s_j\| = 1$.

Note that the ground plane is defined as the xy-plane and its connectivity graph G_{GP} is, by definition, connected. Positions $s \in S_{\text{GP}}$ are not allowed to be occupied by agents, therefore $a_i \in A_i \setminus S_{\text{GP}},\ \forall i \in \mathcal{P}$. Using the graph G_{GP}, we define $G' = (V', E')$ as $V' = V \cup V_{\text{GP}}$ and $E' = V' \times V'$ such that $e_{ij} \in E'$ if for $v_i, v_j \in V'$ we have $\|a_i - a_j\|_{L_1} = 1$. Note that G' represents the current configuration including the ground plane, and a_i represents an action of an agent or an unoccupied position in the ground plane.

Definition 28 (Groundedness)**.** *An agent* i *is grounded if there exists a path on* G' *from* $v_i \in V \subset V'$ *to some* $v_k \in V_{GP} \subset V'$.[9] *A configuration* $\mathcal{C}$ *is grounded if every agent* $i \in \mathcal{P}$ *is grounded.*

The idea behind groundedness hints at an embedding of a self-reconfigurable system in the physical world, where agents cannot choose arbitrary positions in the environment. Typically, agents are prevented from disconnecting from the rest of the configuration through connectivity maintenance (for example [30, 60]). Connectivity, however, is a global property that is computationally costly to enforce and requires the configuration to move as one

[9] *Here,* v_i *represents agent* i *in the connectivity graph* G *(see Def. 23) and* v_k *represents an unoccupied ground plane position* $s \in S_{GP}$ *in the connectivity graph* G_{GP}.

joint group of modules. Although connectivity does enforce cohesion among agents, it is a rather inflexible property as it does not allow splitting and merging of (subsets of) agents. Groundedness, like connectivity, also enforces some level of cohesion (since it prevents floating agents), but without incurring computationally expensive global connectivity checks. One can think of a grounded configuration as one that remains globally connected through the ground plane. In addition, we can use the notion of groundedness to prove completeness of deterministic reconfiguration in Section 4.2.3.

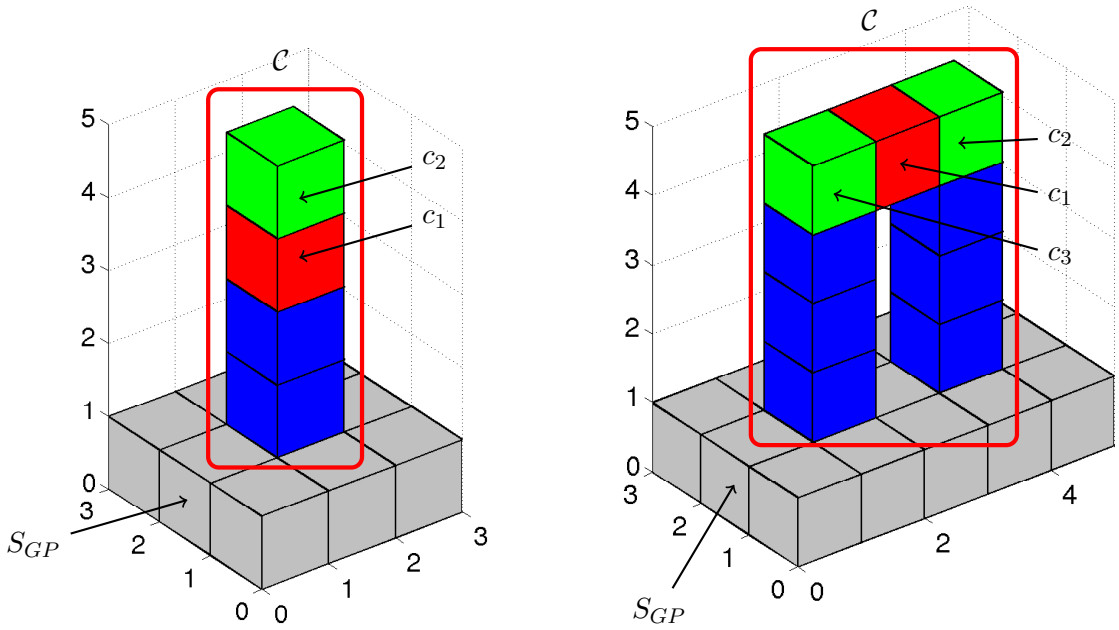
An agent can verify groundedness in a computationally inexpensive way through a depth-first search, which is complete and guaranteed to terminate in time proportional to $O(N)$ in a finite state space. The notion of groundedness also informs the restricted action set computation. If all of agent i's neighbors $\mathcal{N}_i = \{v_j \in V : e_{ij} \in E\}$ (see Def. 23) can compute an alternate path to ground (other than through agent i) then agent i is allowed to move. To formalize this idea, let $G_{-i} = (V_{-i}, E_{-i})$ with $V_{-i} = V \cup V_{\mathrm{GP}} \setminus \{v_i\}$ and $E_{-i} = V_{-i} \times V_{-i}$ such that $e_{ij} \in E_{-i}$ if for $v_i, v_j \in V_{-i}$ we have $\|a_i - a_j\|_{L_1} = 1$. G_{-i} is therefore the connectivity graph of the current configuration including the ground plane without agent i. $R_i^{3\mathrm{D}}(a)$ is then defined as follows.

Definition 29. *The three-dimensional restricted action set $R_i^{3D}(a) = \mathcal{M}_s \cup \mathcal{M}_c$ if all agents $v_j \in \mathcal{N}_i$ are grounded on G_{-i}. Otherwise, $R_i^{3D}(a) = \{a_i\}$.*

This definition encodes the same criteria as the two-dimensional action set with the additional constraint of maintaining groundedness (an example is shown in Fig. 22). If agent i executing an action would leave any of its neighbors ungrounded, agent i is not allowed to move.

4.2.3 Deterministic Completeness

In this section we establish the completeness of deterministic reconfiguration in two and three dimensions. We show that for any two configurations $\mathcal{C}_I$ and $\mathcal{C}_T$ there exists a deterministically determined sequence of individual agent actions such that configuration $\mathcal{C}_I$ will be reconfigured into $\mathcal{C}_T$. These results are required to show irreducibility of the Markov chain induced by the learning algorithm outlined in Section 4.2.4. Irreducibility guarantees



(a) Movement of agent c_1 would remove groundedness of agent c_2.

(b) Agent c_1 can move without breaking the groundedness constraint for agents c_2 and c_3.

Figure 22: Examples of grounded configurations and feasible motions of cubes.

the existence of a unique stationary distribution and furthermore a unique potential function maximizer. The results below have also been presented in [144, 147].

Theorem 7 (Completeness of 2D reconfiguration)**.** *Any given two-dimensional configuration $\mathcal{C}_I$ can be reconfigured into any other two-dimensional configuration $\mathcal{C}_T$, i.e. there exists a finite sequence of configurations $\{\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_M\}$ (with $\mathcal{C}_0 = \mathcal{C}_I$ and $\mathcal{C}_M = \mathcal{C}_T$) such that two consecutive configurations differ only in one individual agent motion.*

Proof. Without loss of generality, assume that $\mathcal{C}_I$ and $\mathcal{C}_T$ do not overlap, i.e. for no $c_i \in \mathcal{C}_I$ is it true that also $c_i \in \mathcal{C}_T$. Additionally, assume that $\mathcal{C}_I$ and $\mathcal{C}_T$ are separated along one dimension $k \in \{b_x, b_y, b_z\}$ (with b_x, b_y, b_z being the basis vectors of the lattice), i.e. $\forall c_{i,k} \in \mathcal{C}_I$ we have that $c_{i,k} < c_{j,k}$, $\forall c_j \in \mathcal{C}_T$. Then at each time step t, select the agent i whose current position $c_i \in \mathcal{C}$ is closest to an unoccupied position $c_j \in \mathcal{C}_T$. Plan a deterministic path of primitive agent motions $p_i = \{a_i^0, a_i^1, \ldots, a_i^m\}$ from an initial module position $a_i^0 = a_i$ to a target position $a_i^m = a_{t_i}$ using a complete path planner such as A*. Note that such a path always exists since we do not require agents to remain connected to any other agents.

Therefore, the path planning problem is reduced to single agent path planning on a discrete finite grid, which is complete because A* is complete. This greedy selection process of the agent-target pairs together with a complete path planning approach suffices to reconfigure any two-dimensional configuration into any other two-dimensional configuration. □

The result in Theorem 7 holds for any configuration, even configurations that consist of multiple connected components. A similar result can be shown for three-dimensional reconfiguration as follows.

Theorem 8 (Completeness of 3D to 2D reconfiguration)**.** *Any finite grounded 3D configuration $\mathcal{C}^{G,3D}$ can be reconfigured into a 2D configuration $\mathcal{C}^{2D}_{Int}$, i.e. there exists a finite sequence of configurations $\{\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_M\}$ (with $\mathcal{C}_0 = \mathcal{C}^{G,3D}$ and $\mathcal{C}_M = \mathcal{C}^{2D}_{Int}$) such that two consecutive configurations differ only in one individual agent motion.*

Proof. Without loss of generality, assume that the connectivity graph of $\mathcal{C}^{\mathrm{G,3D}}$ consists of one connected component. In any finite grounded 3D configuration, there always exists an agent $i \in \mathcal{P}$ with a nonempty restricted action set $|R_i(a)| > 0$. Agent i is therefore mobile and there exists a finite path of individual agent motions $p_i = \{a_i^0, a_i^1, \ldots, a_i^m\}$ (with $a_i^0 = a_i$ being the agent's current action) such that for some $s \in S_{\mathrm{GP}}$, $\|a_i^m, s\|_{L_1} = 1$, i.e. the agent's final action is a position on the ground plane S_{GP}.

Let the subset of agents $\mathcal{P}_{z>1} \subset \mathcal{P}$ contain those agents whose positions are not on the ground plane and $G_{z>1} = (V_{z>1}, E_{z>1})$ the corresponding connectivity graph. Furthermore, let $G'' = (V'', E'')$ be such that $V'' = V_{z>1} \cup \{v^{\mathrm{GP}}\}$, where v^{GP} is a single node representing all the agents on the ground plane and E'' such that $e_{ij} \in E''$ if for $v_i, v_j \in V''$ we have $\|a_i - a_j\|_{L_1} = 1$. According to Def. 29, an agent i is mobile if it is not an articulation point in the connectivity graph G'' (see Fig. 23). In G'', v^{GP} may or may not be a non-articulation points, but according to Lemma 1, in every connected graph there always exist at least two non-articulation points. Therefore, there exists at least one agent i that has a nonempty restricted action set. For that agent, one can compute a deterministic action sequence that moves agent i to the ground plane. Observe that agent i remains mobile on its action path since it is the only agent moving. In other words, at each iteration t we transfer one vertex

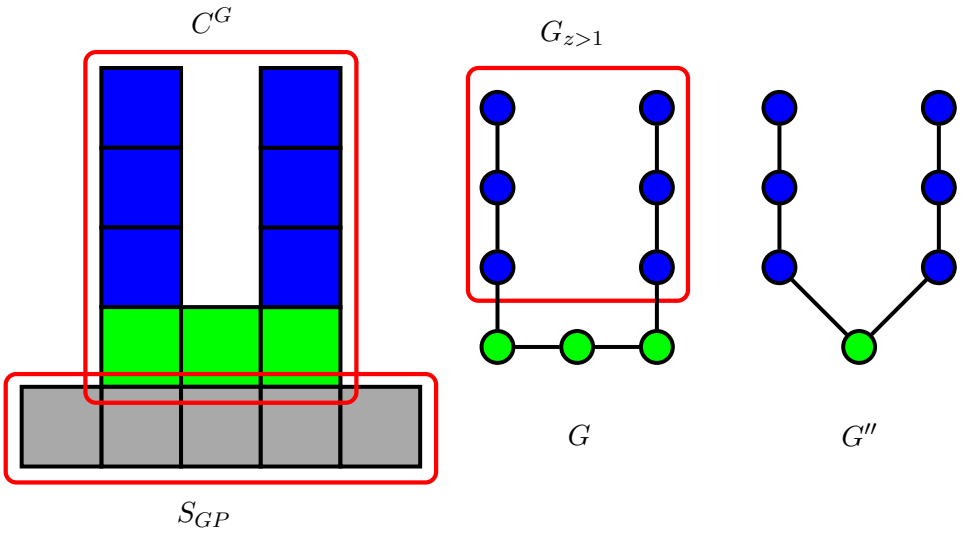


Figure 23: Example of a grounded configuration $\mathcal{C}^G$, the ground plane S_{GP}, associated connectivity graph G. $G_{z>1}$ represents all agents not on the ground plane, while all agents on the ground plane are represented by a single node in G''.

from $V_{z>1}$ to v^{GP} such that $|V^t_{z>1}| = |V^{t-1}_{z>1}| - 1$ until $|V_{z>1}| = 0$ and all agents have been moved to v^{GP}. This process terminates in a finite number of time steps because the initial configuration $\mathcal{C}^{\text{G,3D}}$ is finite. The result is a 2D configuration $\mathcal{C}^{\text{2D}}_{\text{Int}}$ representing G''. □

Corollary 1. *Any finite grounded 3D configuration $\mathcal{C}_I^{G,3D}$ can be reconfigured into any other finite grounded 3D configuration $\mathcal{C}_T^{G,3D}$.*

Proof. Since, according to Theorem 8, any finite grounded 3D configuration $\mathcal{C}^{\text{G,3D}}_I$ can be reduced to an intermediate 2D configuration $\mathcal{C}^{\text{2D}}_{\text{Int}}$ in a finite number of steps, the reverse is also true - any finite grounded 3D configuration $\mathcal{C}^{\text{G,3D}}_T$ can be assembled from some 2D configuration $\mathcal{C}^{\text{2D}'}_{\text{Int}}$ in a finite number of steps. According to Theorem 7, any 2D configuration $\mathcal{C}^{\text{2D}}_{\text{Int}}$ can be reconfigured into any other 2D configuration $\mathcal{C}^{\text{2D}'}_{\text{Int}}$. Therefore, there exists a deterministic finite action sequence from $\mathcal{C}^{\text{G,3D}}_I$ to $\mathcal{C}^{\text{G,3D}}_T$. □

4.2.4 Stochastic Reconfiguration

Building on the problem formulation in Section 4.2.1 and the completeness results in the previous section, in this and the following section we introduce a stochastic reconfiguration algorithm to solve the game-theoretic self-reconfiguration problem. The presented algorithm

is fully distributed, does not require any precomputation of paths or actions, and can adapt to changing environment conditions. Problems that can be formulated as game-theoretic coverage problems are often solved using methods like log-linear learning (LLL, [15]) or variants such as binary log-linear learning (BLLL, see [7, 114, 117, 118]). However, neither can handle the general constraints imposed by the motion and collision constraints of the sliding cube model. LLL is not capable of handling restricted action sets at all while BLLL can only handle action sets constrained by an agent's own previous action. Unlike LLL and BLLL, the algorithm presented in this section guarantees convergence to the potential function maximizer even if action sets are constrained by all (or a subset of other) agents' actions.

Our algorithm is based on the Metropolis-Hastings algorithm ([80, 124]), which allows the design of transition probabilities such that the stationary distribution of the underlying Markov chain is a desired target distribution. In this work the target distribution was chosen to be the Gibbs distribution in accordance with the log-linear learning literature. This choice enables a distributed implementation of the learning rule in Theorem 9 through the potential game formalism (see Corollary 2). The Metropolis-Hastings algorithm guarantees two results: the existence and the uniqueness of a stationary distribution. We will use these properties to show that the only *stochastically stable state* (see Def. 30) is x^*, the potential function maximizer. The following theorem provides a precise formulation of the desired target distribution and demonstrates how to construct transition probabilities that guarantee convergence to the target distribution.

Theorem 9. *Given any two states x_i and x_j representing global configurations and a learning rate $\tau > 0$, the transition probabilities*[10]

$$p_{ij} = \begin{cases} q_{ji} e^{\frac{1}{\tau}(\Phi(x_j)-\Phi(x_i))} & \text{if } e^{\frac{1}{\tau}(\Phi(x_j)-\Phi(x_i))} \frac{q_{ji}}{q_{ij}} \leq 1 \\ q_{ij} & \text{o.w.} \end{cases} \tag{17}$$

guarantee that the unique stationary distribution of the underlying Markov chain is a Gibbs

[10] *Note that $q_{ij} = \frac{1}{|R_k|}$, i.e. a random choice among all available actions in the restricted action set R_k of agent k.*

distribution of the form $Pr[X = x] = \frac{e^{\frac{1}{\tau}\Phi(x)}}{\sum_{x' \in \mathcal{X}} e^{\frac{1}{\tau}\Phi(x')}}$.

Proof. Let $\mathcal{X}$ be a finite state space containing all possible states of configurations composed of N agents.[11] On that state space, let the desired target distribution be $\pi(x) = Pr[X = x] = \frac{e^{\frac{1}{\tau}\Phi(x)}}{\sum_{x' \in \mathcal{X}} e^{\frac{1}{\tau}\Phi(x')}}$ with Φ defined in Def. 22. By applying the Metropolis-Hastings algorithm, we can compute transition probabilities $P = \{p_{ij}\}$ such that π is the stationary distribution of P, i.e. $\pi = \pi P$. In the Metropolis-Hastings algorithm, a transition probability is represented as $p_{ij} = g(x_i \to x_j)\alpha(x_i \to x_j)$, where $g(x_i \to x_j)$ is the *proposal distribution* and $\alpha(x_i \to x_j)$ is the *acceptance distribution*. Both are conditional probabilities of proposing/accepting a state x_j given that the current state is x_i.

Let agent k achieve the transition from state x_i to x_j through action $a_k \in \mathcal{R}_k(a)$. Then one possible choice for the proposal distribution is $g(x_i \to x_j) = q_{ij} = \frac{1}{|R_k|}$, $\forall j \in \{1, \ldots, |R_k|\}$, i.e. a random choice among all available actions of agent k. According to Hastings ([80]), a popular choice for the acceptance distribution is the Metropolis choice $\alpha_{ij} = \min\left\{1, \frac{\pi_j q_{ji}}{\pi_i q_{ij}}\right\}$. Note that unlike in the original formulation ([124]), we do not assume symmetric proposal probabilities, i.e. $q_{ij} = q_{ji}$ (see Fig. 24 for an illustration of q_{ij} and q_{ji}). These choices result in the following transition probabilities:

$$p_{ij} = \begin{cases} q_{ji}\frac{\pi_j}{\pi_i} & \text{if } \frac{\pi_j q_{ji}}{\pi_i q_{ij}} \leq 1 \\ q_{ij} & \text{o.w.} \end{cases} \tag{18}$$

It is easily verified that these p_{ij} satisfy the detailed balance equation $\pi_i p_{ij} = \pi_j p_{ji}$ and thus guarantee the existence of a stationary distribution (see [80, 124]). The resulting p_{ij} follow from the definition of $\pi_i = \frac{e^{\frac{1}{\tau}\Phi(x_i)}}{\sum_{x' \in \mathcal{X}} e^{\frac{1}{\tau}\Phi(x')}}$ and similarly π_j. Uniqueness of the stationary distribution follows from the irreducibility of the Markov chain induced by $P = \{p_{ij}\}$, which is the case for our choice of proposal and acceptance distribution because they assign a nonzero probability to every action in any restricted action set. By Theorem 8 and Corollary 1 we know that any state x_j can be reached from any other state x_i and vice versa. Thus any action path has nonzero probability and irreducibility follows. □

[11] Such a space is finite if we assume a finite environment.

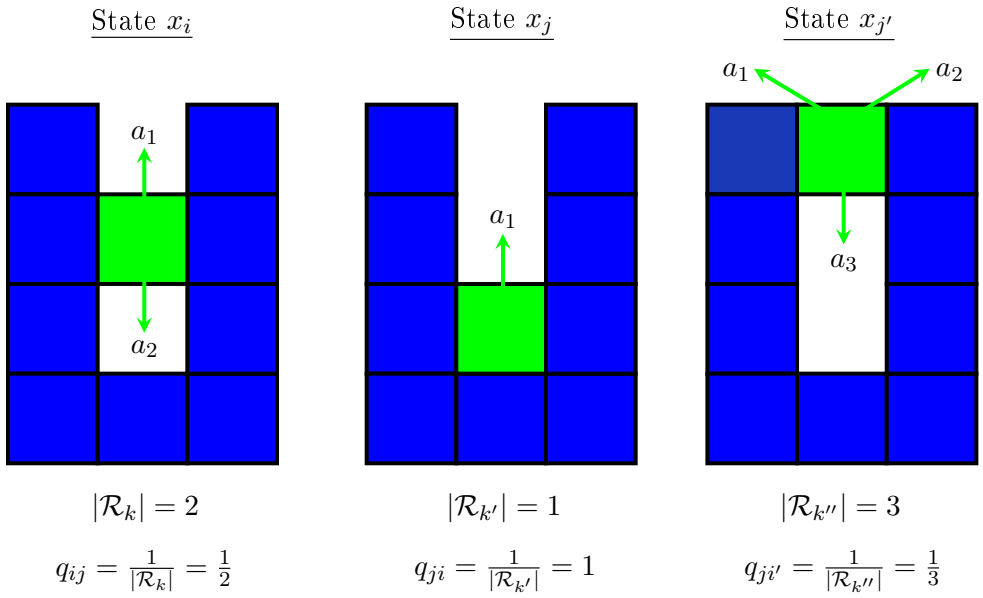


Figure 24: Example of forward and reverse actions with their associated proposal probabilities q_{ij}, q_{ji}, and q'_{ji}. Note that x_i, x_j, x'_j are states of the entire configuration, and agent k is the currently active agent.

Theorem 9 applies equally for 2D and 3D configuration. However, for 3D reconfiguration, the proof relies implicitly on the notion of groundedness to show irreducibility of the underlying Markov chain (through the computation of $R_i^{3\mathrm{D}}$). The following theorem requires the definition of stochastic stability.

Definition 30 (Stochastic Stability [66, 197, 198])**.** *A state $x_i \in \mathcal{X}$ is stochastically stable relative to a regular perturbed Markov process P^ϵ with noise parameter $\epsilon > 0$ if the following holds for the stationary distribution π.*

$$\lim_{\epsilon \to 0^+} \pi^\epsilon_{x_i} > 0 \tag{19}$$

Note that the Markov process is defined through the transition probabilities in Theorem 9 and the stationary distribution is a Gibbs distribution. Furthermore ϵ is equivalent to the learning rate τ in the following proof.

Theorem 10. *Consider the self-reconfiguration problem in Def. 22. If all players adhere to the learning rule in Theorem 9 then the unique stochastically stable state x^* is the state that*

maximizes the global potential function.

Proof. Note that this result holds by definition of the desired target distribution, which is a Gibbs distribution centered at the state of maximum global potential and defines the probability of being in a state x as $Pr[X = x] = \frac{e^{\frac{1}{\tau}\Phi(x)}}{\sum_{x' \in \mathcal{X}} e^{\frac{1}{\tau}\Phi(x')}}$. The learning rate or temperature τ represents the willingness of an agent to make a suboptimal choice (i.e. explore the state space). As $\tau \to 0$, $Pr[X = x] \to 0$ for all states $x \neq x^*$ which are not potential function maximizers (see [15, 118]). Since the global potential as well as the desired target distribution have a unique maximizer, the set of stochastically stable states contains a single, unique state x^*. □

Note that the maximum global potential is achieved when all agents are at a target position $a_i \in \mathcal{C}_T$. Algorithm 4 shows an implementation of Theorem 9. In Algorithm 4, similarly to the Metropolis-Hastings algorithm in [80], $p_{ii} = 1 - \sum_{j \neq i} p_{ij}$.

Algorithm 4 Centralized game-theoretic learning algorithm. Note that state x_j is the result of agent k applying action $a_{i \to j,k}$ and x_i and x_j refer to states of the entire configuration.

Require: Current and target configuration $\mathcal{C}$ and $\mathcal{C}_T$
1: **while** True **do**
2: Randomly pick an agent k in state x_i
3: Compute restricted action set $\mathcal{R}_k$
4: Select $a_{i \to j,k} \in \mathcal{R}_k$ with probability $q_{ij} = \frac{1}{|\mathcal{R}_k|}$
5: Compute $\alpha_{ij} = \min\left\{1, \frac{q_{ji}}{q_{ij}} e^{\frac{1}{\tau}(\Phi(x_j) - \Phi(x_i))}\right\}$
6: **if** $\alpha_{ij} == 1$ **then**
7: $x_{t+1} = x_j$
8: **else**
9: $x_{t+1} = \begin{cases} x_j & \text{with probability } \alpha_{ij} \\ x_i & \text{with probability } 1 - \alpha_{ij} \end{cases}$
10: **end if**
11: **end while**

4.2.5 A decentralized Algorithm

Algorithm 4 shows a centralized implementation of the game-theoretic learning method of Theorem 9 which requires the computation of a global potential function $\Phi(x_i)$ and depends on the current joint action x_i (i.e. the entire current configuration $\mathcal{C}_I$).[12] A decentralized

[12] Note that a global state of a configuration x_i and a joint action a are used interchangeably.

algorithm, however, is desirable to execute the learning rule of Theorem 9 on a team of agents without a central coordinator. The formulation of the self-reconfiguration problem as a potential game allows us to rewrite the transition probabilities in a decentralized fashion as follows.

Corollary 2. *The global learning rule of Theorem 9 can be decentralized through locally computable transition probabilities p_{ij}, where*

$$p_{ij} = \begin{cases} q_{ji} e^{\frac{1}{\tau}(U_k(a'_k) - U_k(a_k))} & \text{if } e^{\frac{1}{\tau}(U_k(a'_k) - U(a_k))} \frac{q_{ji}}{q_{ij}} \leq 1 \\ q_{ij} & \text{o.w.} \end{cases} \tag{20}$$

Proof. Note that for agent k, we can express the global states or joint actions x_j and x_i as (a'_k, a_{-k}) and (a_k, a_{-k}) respectively. According to Proposition 1, we can then rewrite $\Phi(x_j) - \Phi(x_i)$ as follows.

$$\begin{aligned} \Phi(x_j) - \Phi(x_i) &= \Phi(a'_k, a_{-k}) - \Phi(a_k, a_{-k}) \\ &= U_k(a'_k, a_{-k}) - U_k(a_k, a_{-k}) \\ &= U_k(a'_k) - U_k(a_k) \end{aligned}$$

The last step in this rewriting relies on the fact that an agent's individual utility function depends only on its own action and not the action of any other agents. The resulting transition probabilities p_{ij} then follow from Theorem 9. Since q_{ij}, q_{ji}, $U_k(a'_k)$, as well as $U_k(a_k)$ can be computed with local information, so too can the transition probabilities. The stationary distribution of the Markov process described by p_{ij} is the same Gibbs distribution as in Theorem 9. □

Note that local information in this context can mean multiple hops, because the computation of restricted action sets requires the maintenance of groundedness of all neighboring agents. Verifying groundedness requires $N - 1$ hops in the worst case (for the case of a line configuration of agents).

Algorithm 5 shows a decentralized implementation of Algorithm 4 and Corollary 2. Note that a transition from configuration x_i to x_j is accomplished by agent k executing action

$a'_k \in \mathcal{R}(a_k, a_{-k})$ based on its current position or action a_k. Therefore, we can interpret q_{ij} as the transition probability of the forward action and q_{ji} as that of the reverse action (see Fig. 24). Also note that any action is always reversible, i.e. for any action $a'_k \in \mathcal{R}(a_k, a_{-k})$ it is always true that $a_k \in \mathcal{R}(a'_k, a_{-k})$. It is therefore always possible to revert back to a_k after executing an action $a'_k \in \mathcal{R}(a_k, a_{-k})$. Note that a_{-k} in this section only refers to agents in the local neighborhood of agent k and not all agents in the configuration. In Algorithm 5, decentralization is achieved by forcing agents to take turns in executing actions. Each agent uses a specific type of clock that ticks according to a rate 1 Poisson process. According to [22], on average only one clock ticks per time step, meaning that, on average, only one agent is active and moves at a time.[13]

Algorithm 5 Decentralized self-reconfiguration using game-theoretic learning, that each agent can executes with local information. Note that x_t, x_{t+1} refer to consecutive states of the active agent.

Require: Target configuration $\mathcal{C}_T$

Start clock (see [22])
while True **do**
  if Clock ticks **then**
    Compute current restricted action set $\mathcal{R}_k$
    Select $a'_k \in \mathcal{R}(a_k, a_{-k})$ with probability $q = \frac{1}{|\mathcal{R}(a_k, a_{-k})|}$
    Compute $\alpha = \min\left\{1, \frac{|\mathcal{R}(a_k, a_{-k})|}{|\mathcal{R}(a'_k, a_{-k})|} e^{\frac{1}{\tau}(U(a') - U(a))}\right\}$
    if $\alpha == 1$ **then**
      $x_{t+1} = a'$
    else
      $$x_{t+1} = \begin{cases} a' & \text{with probability } \alpha \\ a & \text{with probability } 1 - \alpha \end{cases}$$
    end if
  end if
end while

4.2.6 Results

Algorithm 5 was implemented and evaluated in Matlab with a learning rate $\tau = 0.001$ that struck a balance between greedy maximization of agent utilities and exploration of the state space through suboptimal actions. Agents' restricted action sets depended on the agents'

[13] If more than one agent is active in the same local neighborhood, a decentralized blocking mechanism is used to ensure the sequential execution of actions.

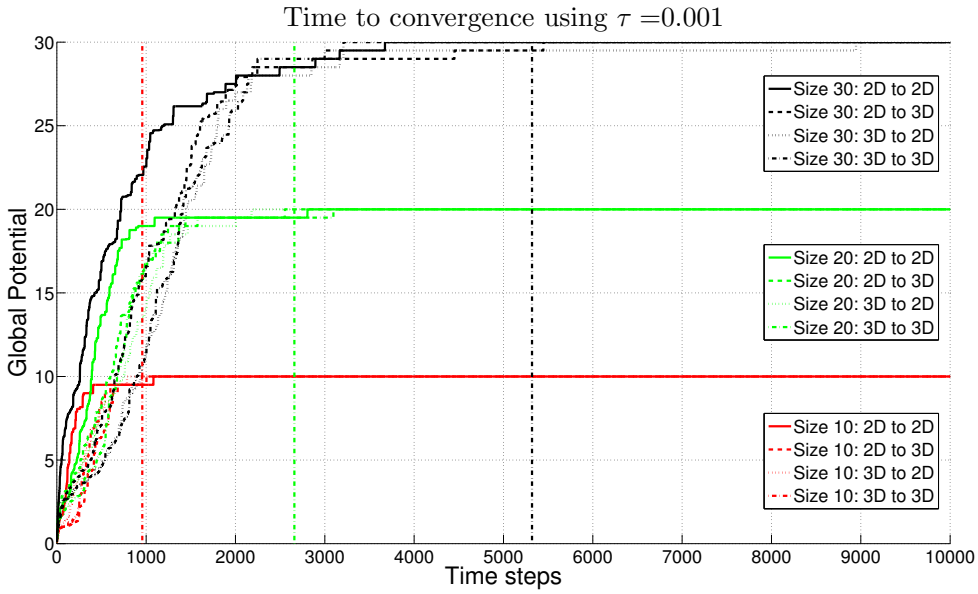
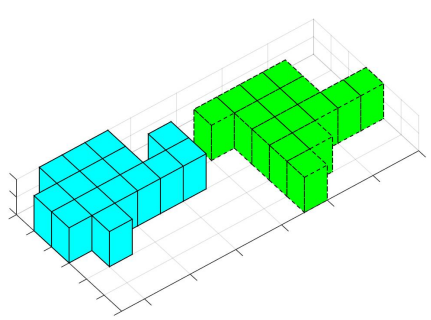


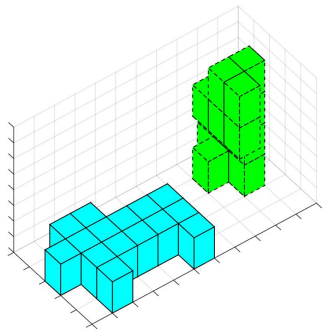
Figure 25: Convergence times for different types of configurations and sizes ranging from 10 to 30 agents. A fixed learning rate $\tau = 0.001$ was used for the shown results.

joint action and the environment - in these simulations only the ground plane (agents were initialized on or above the ground plane, i.e. their z-coordinate $z \geq 1$). In a straightforward extension to this algorithm, obstacles can be added to restrict the environment even further.

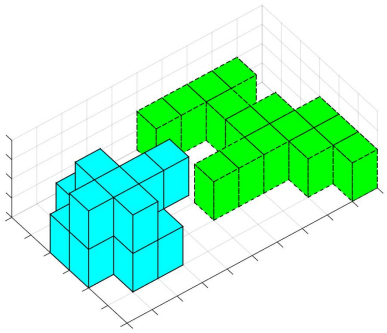
Fig. 25 shows convergence results of Algorithm 4 of configurations containing 10, 20, and 30 agents. Four types of reconfigurations have been performed: 2D to 2D, 2D to 3D, 3D to 2D, and 3D to 3D (see Fig. 26). Ten trials were conducted for each scenario and convergence was achieved once the configuration reached a global potential of $\Phi = N$, i.e. every agent reached a utility of $U_i = 1$. The vertical lines in Fig. 25 represent the average time to convergence of all four types of reconfigurations of a certain size (e.g. the leftmost line represents the average convergence time of a configuration of 10 agents). Note that for the scenarios of Fig. 25, the target configuration was offset from the initial configuration by a translation of 10 units along the x-axis (this offset is not shown in Fig. 26). One can observe that at the beginning of each reconfiguration the global potential ramps up very fast (within a few hundred time steps), but the asymptotic convergence to the global optimum can be slow (see the case 3D to 2D for 30 agents).



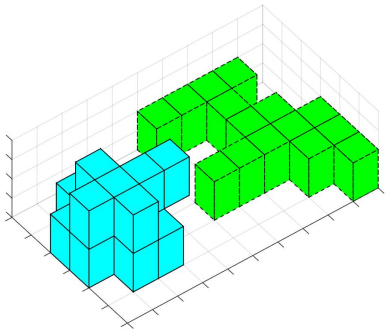
(a) 2D to 2D reconfiguration



(b) 2D to 3D reconfiguration



(c) 3D to 2D reconfiguration



(d) 3D to 3D reconfiguration

Figure 26: Examples of various randomly generated scenarios for each combination of two- and three-dimensional initial and target configurations.

4.3 Conclusions

This chapter introduced two methods for accomplishing decentralized self-reconfiguration. The graph grammar-based method in Section 4.1 relies on sets of paths as input and precomputes a graph grammar, which can then be executed on a set of agents in a distributed fashion. While allowing a decentralized execution, this method relies on precomputation as well as messages being broadcast to the entire configuration.

The game-theoretic method presented in Section 4.2 alleviates these issues and adds a number of appealing properties to a novel self-reconfiguration approach. First, we formulated the self-reconfiguration problem as a potential game with each cube being modeled as an autonomous decision maker or agent. Each agent is capable of determining its own action set together with transition probabilities for each action. Actions are chosen based on local information but guarantee the assembly of the global target configuration. This property is guaranteed by Theorem 9 and Theorem 10 through the computation of transition probabilities which are based on the Metropolis-Hastings algorithm. The resulting decentralized self-reconfiguration algorithm (Algorithm 5) then requires only local information for decision making, does not rely on precomputation, uses no (in the 2D case) or limited communication (in the 3D case), can assembly arbitrary two- or three-dimensional configurations, and is guaranteed to converge. Additionally, it can adapt during runtime to changes in the environment, the target configuration, or even the number of agents part of the configuration. However, the main contribution of the presented algorithm is the fact that it guarantees convergence despite the fact that agents can constrain each other. Neither of the often used algorithms - log-linear learning and binary log-linear learning - is capable of handling this type of constraint.

The next chapter will focus on extensions to the basic algorithms outlined in Section 4.2 with the goal of improving convergence rates. Specifically, Chapter 5 investigates extensions that are capable of adjusting the learning rate or the utility function used in an agent's learning rule during runtime. The goal is to equip the basic game-theoretic learning rule with a certain degree of adaptability to changing or unknown environments while retaining convergence guarantees.

Chapter V

ADAPTIVE DECENTRALIZED METHODS

The game-theoretic learning method introduced in Section 4.2 relies on a fixed rule for computing transition probabilities for available agent actions. While these probabilities can vary depending on an agent's immediate surroundings and its restricted action set, the decision making rule remains fixed. As we have seen in Section 4.2.6, fixed learning rates τ or fixed utility functions can lead to fast convergence times in specific scenarios but perform poorly in others. For example, environments containing obstacles typically favor larger values for τ or utility functions with built-in obstacle avoidance. However, the assembly of target configurations in obstacle-free environments typically converges faster when using low values of τ or utility functions geared towards assembly. Since no given fixed learning rule fits all scenarios, the performance of game-theoretic self-reconfiguration will benefit from a certain level of adaptability in the learning algorithms.

Therefore, in this section we will outline two approaches that make the learning rules of Algorithm 4 and 5 tunable at runtime with the goal of improving convergence times to the target configuration. The computation of transition probabilities allows two approaches to runtime adaptation - modifying an agent's learning rate τ (which changes an agent's willingness to explore or exploit) or modifying an agent's utility function (giving it essentially a different objective to pursue). We will first discuss an adaptive learning rate method in Section 5.1 and proceed by introducing a method for switching utility functions in Section 5.2. More specifically, Section 5.2 presents a framework for creating finite state machines that control the utility switching of individual agents. Simulation results for both methods (shown in Section 5.1.2 and 5.2.4) indicate improved performance over the basic approach in Section 4.2 for non-trivial scenarios. The main theoretic contributions of this section provide convergence guarantees for both adaptive methods and prove that a given target configuration will indeed be assembled.

5.1 Adaptive Learning Rate

As previous results indicate (see Section 4.2.6 and [144, 147]), fixed values of τ in the learning rule of Theorem 9 can lead to slow convergence for both small and large values of τ. Fixed, large values of τ lead to rapid exploration, which is advantageous when obstacles are present in the environment. However, such agent behavior is typically ill-suited for finalizing the assembly of $\mathcal{C}_T$ because of its propensity to explore the state space. Inferior actions are therefore more readily accepted and agent motions tend to be more volatile. Small values of τ, on the other hand, lead to greedy exploitation and a rapid approach of $\mathcal{C}_T$. However, self-reconfiguration with small values of τ tend to take a large number of iterations to overcome obstacles because the probability of accepting inferior actions is small. In general, neither small nor large values of τ lead to low convergence times but the performance for a fixed value is highly scenario-dependent. In this section we therefore outline a strategy for adapting the value of τ on the fly and on a per-agent basis meaning that different agents can use different values of τ at any given time.

The adaptation strategy in this section is based on the idea that by default, an agent uses a small nominal value of $\tau = \tau_{\text{nom}}$ (i.e. greedily exploits) and temporarily increases its own learning rate to $\tau_{\max}$ if no improvement in utility can be achieved within N iterations (for the results shown, we chose $N = 15$).[1] An increase in τ increases an agent's probability of accepting inferior actions and thus its willingness to explore. However, to maintain a notion of cohesion among agents and affinity to $\mathcal{C}_T$, τ undergoes an exponential decay back to its nominal value τ_{nom}. Figure 27 shows an example of this adaptive-τ-strategy where discontinuous jumps in the value of τ can be seen after an agent executes actions for N iterations without an improvement in utility.

As shown in Fig. 28 this time-varying trade-off between exploration (large values of τ) and exploitation (small values of τ) improves convergence times in randomly generated environments containing obstacles. Note that this adaptation strategy is just one example of

[1] Regarding large values of τ, note that as $\tau \to \infty$ the performance of the self-reconfiguration process deteriorates to a naive random walk in the state space of the underlying Markov chain. Therefore values of $\tau_{\max}$ should be chosen conservatively to avoid poor performance. In simulation, values of $\tau_{\max} \in [1, 10]$ have shown good performance.

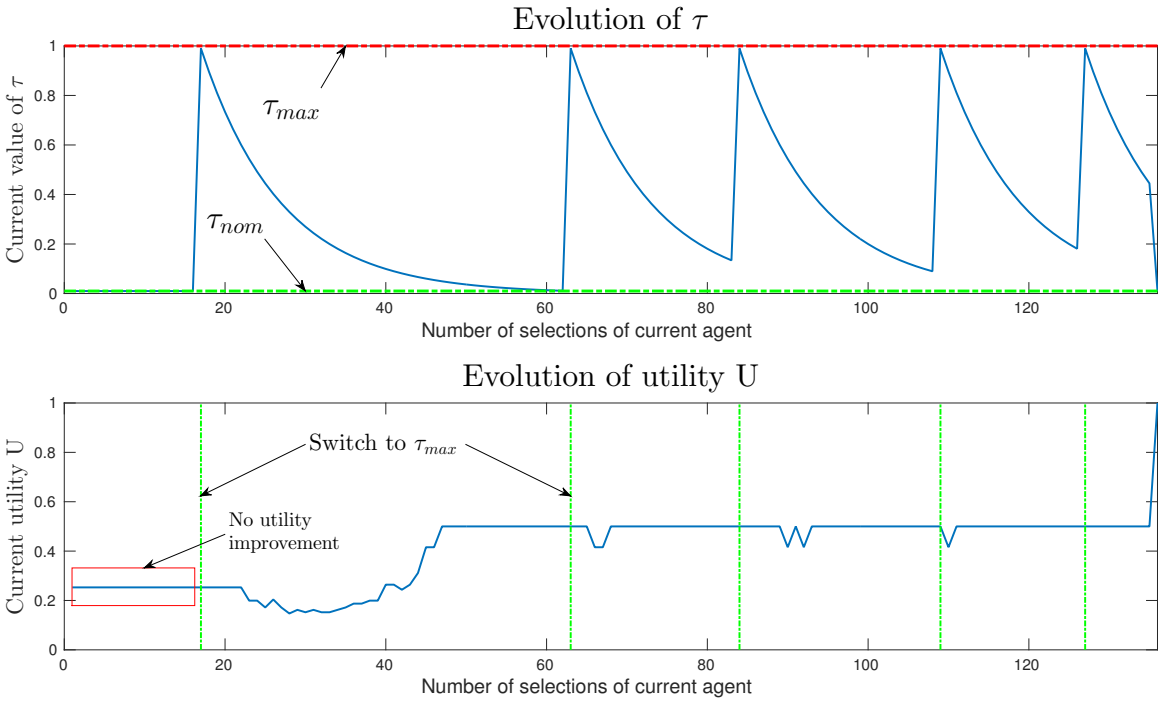


Figure 27: Adaptation strategy for τ with $\tau_{nom} = 0.01$, $\tau_{max} = 1.0$, and $N = 15$. Shown is an example of a utility function time series for a single agent together with a time series of the corresponding learning rate τ.

how the value of τ can be adapted to improve the performance of the learning rule of Theorem 9. For Theorem 10 to hold, it is required that the time-inhomogeneous Markov process (caused by agents adapting their value of τ and thus modifying their transition probabilities) eventually settles into a time-homogeneous Markov process with constant transition probabilities. Therefore, our approach deactivates an agent's adaptive behavior as soon as it reaches a target position $c_i \in \mathcal{C}_T$, which is guaranteed to happen in a finite state space (even for a naive random walk). Even after deactivating the adaptive behavior, the nominal value of $\tau = \tau_{\text{nom}}$ allows for a sufficient level of exploration of the state space and ensures that the Markov process remains irreducible (as required by Theorem 9). At the same time, it also ensures that agents do not block each other from completing the assembly. As a consequence, Theorem 9 and Theorem 10 still hold for the adaptive behavior.

5.1.1 Influence of the Learning Rate

As shown in Theorem 9, transition probabilities consist of a proposal probability and an acceptance probability for each possible action of an agent. Proposal probabilities are exclusively a function of the size of the restricted action set. Acceptance probabilities for a chosen alternative action a', however, depend on the utility difference between the current and an alternative action. As such, the utility difference is a function of both τ as well as the distance d of action a' to $\mathcal{C}_T$.

An approximation of the influence of the value of τ on the acceptance ratio α_{ij} of inferior actions (those that decrease the utility of an agent) is shown in Table 6. Note that Table 6 is a simplification of the probabilities of Theorem 9 because it examines agents in isolation such that $q_{ij} = q_{ji}$. In other words, an agent's actions are not restricted by other agents and the restricted action sets have the same size for the forward and reverse actions. Regardless, Table 6 highlights a noteworthy trend of the acceptance ratio α_{ij}. An agent is more likely to accept an inferior action as it moves farther away from $\mathcal{C}_T$ (i.e. larger d) and as the value of τ increases. In other words, the performance deteriorates towards a naive random walk with increasing distance to $\mathcal{C}_T$ and increasing τ.[2] For example, a value of $\tau = 1$ in Table 6 indicates an acceptance ratio of inferior actions of 0.606 as $d \to 0$. However, as $d \to \infty$ such an action is accepted with near-certainty of 0.99. On the other hand, for a value of $\tau = 10$, inferior actions are accepted with a probability of 0.95 even as the distance $d \to 0$.

The acceptance ratio as a function of d and τ can be interpreted as the trade-off between exploration of the state space and greedy exploitation (i.e. motion towards $\mathcal{C}_T$). It is conceivable that neither small nor large values of τ are well-suited for all scenarios. As we will show in the next section, the adaptive-τ-strategy outlined in this section combines the rapid exploitation of using small values of τ and the propensity towards exploration of large values of τ and outperforms any single fixed value of τ used in this section.

[2] Note that in a finite state space even a naive random walk of agents will eventually lead to the assembly of the target configuration. However, convergence times will be poor since the size of the state space is exponential in the number of agents.

Table 6: Influence of the learning rate τ on the acceptance of actions with decreasing utility.

τ	α_{ij}	
	$d \to 0$	$d \to \infty$
10	0.95	0.999
1	0.606	0.99
0.1	0.006	0.90
0.01	1.92E-22	0.36
0.001	7E-218	4.5E-5

5.1.2 Results

In this section we compare the performance of the basic decentralized implementation of Algorithm 5 using fixed learning rates τ to an implementation of Algorithm 5 using the adaptive-τ-strategy outlined in this section. Both algorithms were implemented and evaluated in Matlab. Agents' actions were restricted according to the action set computation outlined in Section 4.2.2, the motion model in Section 2.3, the ground plane, as well as obstacles present in the environment. Agents' positions were initialized above the ground plane such that their z-coordinate $z \geq 1$. Both the initial and the target configuration $\mathcal{C}_I$ and $\mathcal{C}_T$ were randomly generated. An example of these random scenarios for both two- and three-dimensional reconfiguration is shown in the insets of Fig. 28a and Fig. 28b respectively.

Fig. 28 shows convergence time results for initial and target configurations containing 20 agents for two- and three-dimensional reconfiguration sequences and values of $\tau \in \{0.01, 0.1, 1.0, \tau_{\text{adaptive}}\}$. Ten trials were conducted for each value of τ for both the two-dimensional and three-dimensional scenarios respectively. The vertical lines in Fig. 28 represent the average time to convergence for varying values of τ (e.g. the leftmost line represents the average convergence time for $\tau = \tau_{\text{adaptive}}$). Convergence is achieved, if the configuration reaches a global potential of $\Phi = N$, i.e. every agent achieves a maximum utility of $U_i = 1$. Trials that did not converge within 10,000 time steps were aborted.

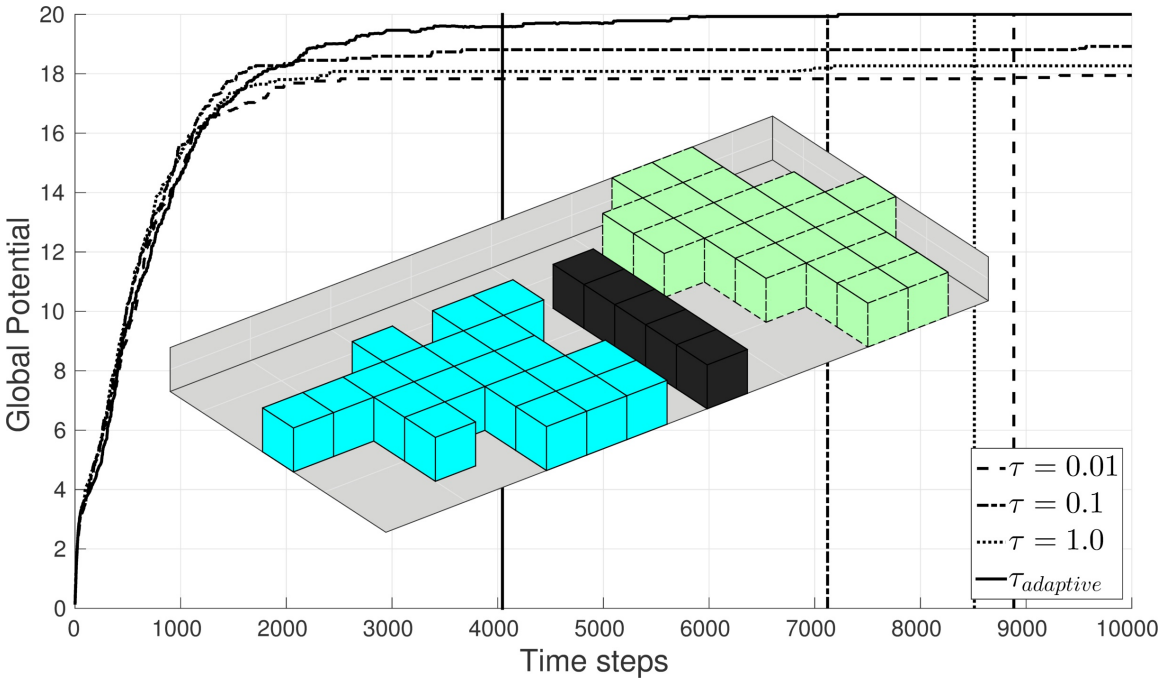
Note that in the scenarios of Fig. 28, the target configuration was offset from the initial configuration by a translation of three units along the x-axis. Additionally, $\mathcal{C}_I$ and $\mathcal{C}_T$ were separated by an obstacle. One can observe that at the beginning of each reconfiguration the

global potential ramps up very fast (within a few hundred time steps), but the asymptotic convergence to the global optimum can be slow or even fail within the chosen time horizon of 10,000 time steps (for certain values of τ and certain scenarios). As seen in Fig. 28, reconfiguration sequences using the adaptive-τ-strategy converged for each of the ten trials. For fixed values of τ, some of the trials did not converge within the allowed time horizon (this can be seen in the figure where the global potential does not reach its maximum value $\Phi = N$). Note however, that convergence to the target configuration is still guaranteed by Theorem 10 (but would require a longer time horizon).

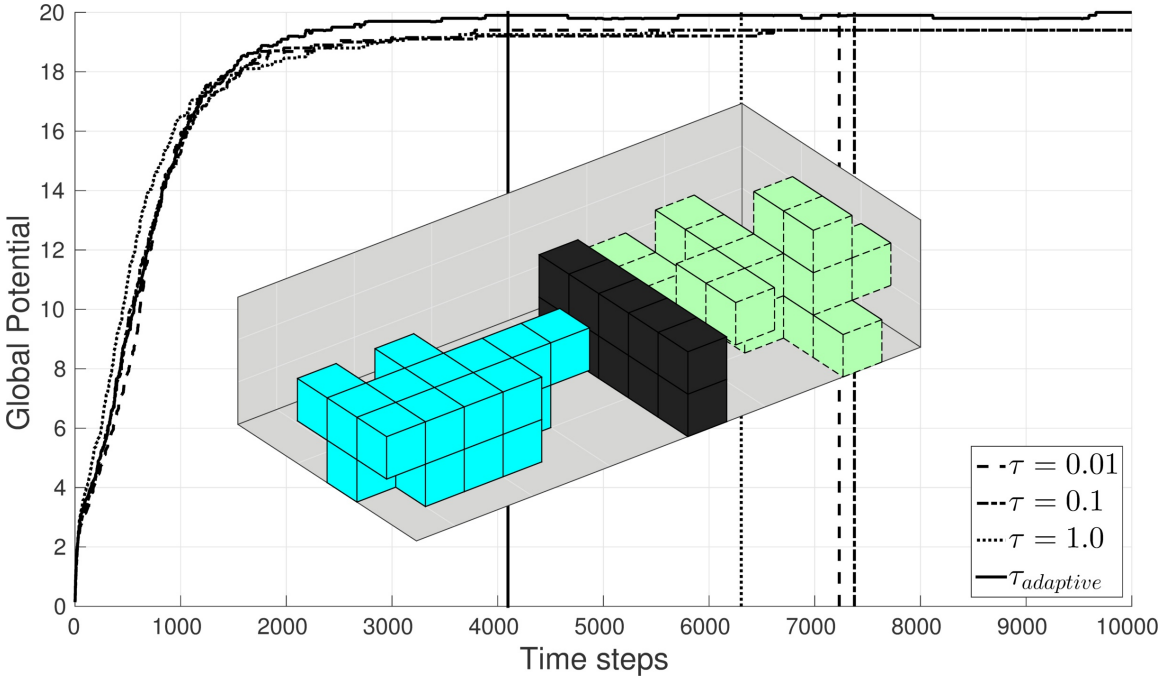
5.2 Switching Utility Functions

Utility function switching is the second approach to runtime adaptation of the learning rule outlined in Section 4.2.4. As opposed to modifying the learning rate τ, in this section we modify the utility function or replace it altogether. In the interest of enabling agents to make utility switching decisions as autonomously as possible, we are interested in an endogenous utility switching approach. In other words, an agent should be capable of determining a switching condition and executing it based on its own state rather than on the global state of the aggregate system or the state of a set of neighbors.

Alternatively, one could focus the decision making capabilities regarding utility function switches in a set of leader agents and set up the problem as a leader-follower network. Leader agents would determine switching conditions, execute switches, and disseminate that information to all followers. While this approach might be advantageous in a setting with asymmetric information (i.e. leaders have access to more information than followers), the setup in this Section guarantees information symmetry because it relies on *state-based switching conditions* (see Def. 31). As such, followers can determine switching conditions based on the same information that leader agents would have access to. In such a setting, leader-based switching would actually remove granularity by forcing followers to switch based on the leader's state as opposed to their own individual state. While this section does not investigate the leader-follower approach any further, Section 5.2.4 presents results on a fully centralized switching approach that could be interpreted as a single leader agent. The



(a) Convergence times for two-dimensional reconfiguration sequences.



(b) Convergence times for three-dimensional reconfiguration sequences.

Figure 28: Convergence times for two- and three-dimensional reconfiguration sequences containing 20 agents for varying values of τ. The results of four different learning rates τ are shown: $\tau = 0.01$, $\tau = 0.1$, $\tau = 1$, and a time-varying τ according to Section 5.1. In both figures, an example of the initial configuration $\mathcal{C}_I$ is shown on the left (in red, opaque), obstacles are shown in the center (in black, opaque), and the target configuration $\mathcal{C}_T$ is shown on the right (in green, semi-transparent).

results in that section suggests that removing granularity actually slows down convergence.

Another alternative is a fully centralized approach that induces switching through exogenous signals such as a broadcast to all agents. This method would require a broadcasting framework or alternatively a global synchronizing signal (for example a magnetic or electric field, or a signal that all agents can measure with onboard sensors). A similar approach has been investigated in [12] in which a self-assembly process is controlled through external force manipulation of particles. While this is a viable approach if a broadcasting signal can be provided to the entire configuration, in this section, we focus on a distributed approach to switching. However, Section 5.2.4 investigates exogenous (centralized) versus endogenous (distributed) utility switching with respect to convergence rates and presents simulation results that suggest faster convergence for endogenous switching. Additionally, this section presents a number of possible utility functions, combines them in linear weighted sums to form new behaviors, and introduces theoretic results that guarantee convergence in the presence of utility function switching.

5.2.1 Utility Function Components

In this section we exclusively present utility functions that depend on an agent's individual state or a measurable state component of the static environment (such as static obstacles). No knowledge about neighboring agents (if any are available) is required for an agent to compute its utility function. The main reason for this is to avoid coupling between the agents through the utility function. Coupling is limited to collision avoidance and groundedness maintenance (in the restricted action set computation). A consequence of decoupled utility functions is that information requirements are lower which reduces delays in the computation of utilities and in the decision making process overall. An agent does not have to gather information about neighboring agents to determine its utility, update its transition probabilities, and then choose an action. All utility function components shown below are limited to the interval $(0, 1]$ (or $[-1, 0)$ for the obstacle-averse utility function) so that their combination as linear weighted sums produces predictable outcomes. Other desirable properties of utility functions are the following.

- Utility functions should satisfies the potential game formalism in Def. 21 to allow a decentralized computation and execution.
- Utility function values should depend exclusively on an agent's own state (or the state of static objects in the environment such as obstacles) to avoid additional coupling between agents beyond collision avoidance and groundedness maintenance.
- The maximum utility value needs to be placed at an action representing a target position (for guaranteeing convergence to the target configuration). Any utility function for which this property does not hold can only be used temporarily (see Theorem 12).

Target configuration-seeking utility This utility function is the default function used for assembling arbitrary structures in Section 4.2. It rewards actions more highly that are closer to the target configuration.

$$U_{\mathrm{T,af}}(d) = \frac{1}{d+1} \text{ or alternatively } U_{\mathrm{T,af}}(d) = e^{-kd} \tag{21}$$

It was already shown in Chapter 4 (Section 4.2) that this utility function satisfies the potential game formalism, which is a requirement for the decentralization of the game-theoretic learning algorithm of Section 4.2.5.

Distance-seeking utility Distance-seeking utility functions are a generalization of target-seeking utility functions in the sense that their maximum is centered at a desired distance to the target configuration. As such, they are formulated as follows.

$$U_{\mathrm{D,af}}(d) = e^{\frac{-(d-d_{\mathrm{des}})^2}{2\sigma^2}} \tag{22}$$

Here, d_{des} is the desired distance to the target configuration and σ is a parameter that determines the standard deviation of the normal distribution. In simulation, values of $\sigma \in [2, 10]$ exhibited good performance. This type of utility function can be used in the construction of utility switching state machines as shown in Section 5.2.

Lemma 4. *The distance-seeking utility function* $U_{D,af} = e^{\frac{-(d-d_{des})^2}{2\sigma^2}}$ *satisfies the potential game formalism according to Definition 21 and Definition 22.*

Proof. This proof follows the same reasoning as shown in Proposition 1 and expresses the global potential Φ as follows.

$$\Phi(a_i, a_{-i}) = \left(e^{\frac{-(d_i - d_{\text{des}})^2}{2\sigma^2}} + \sum_{j \in \mathcal{P} \setminus \{i\}} e^{\frac{-(d_j - d_{\text{des}})^2}{2\sigma^2}} \right)$$

(where d_i is a shorthand notation for $d_i = d(a_i) = \min_{a_j \in \mathcal{C}_T} \|a_i - a_j\|_{L_2}$). The result then follows from the defining equation for potential games $U_i(a_i') - U_i(a_i) = \Phi(a_i', a_{-i}) - \Phi(a_i, a_{-i})$. □

Ground-seeking utility Ground-seeking utility functions reward actions closer to the ground plane more highly. Used by itself, this type of utility function would cause agents to seek positions on the ground plane and essentially "melt" the configuration (similar to an approach shown in [61]). Ground-seeking utility potentially speeds up the reconfiguration from tall three-dimensional initial configurations to other arbitrary two-dimensional or three-dimensional configurations (see Section 5.2.4). It also offers a potential speedup in the locomotion of a configuration.

$$U_{\text{G,af}}(z) = \frac{1}{z+1} \quad \text{or alternatively} \quad U_{\text{G,af}}(z) = e^{-k_z z} \tag{23}$$

One can additionally include a distance-dependent term, which makes agents more ground-seeking the farther they are located from the target configuration as follows.

$$U_{\text{G,af}}(d, z) = (1 - e^{-kd})\frac{1}{z+1} \quad \text{or alternatively} \quad U_{\text{G,af}}(d, z) = (1 - e^{-kd})e^{-k_z z} \tag{24}$$

The rationale behind this distance-dependency is that high agent positions are more likely to slow down convergence when agents are located farther from the target configuration. Simulation suggest that this behavior can indeed slow down convergence through the formation of dendrite-like sub-structures as shown in Fig. 31. Two gain parameters need to be chosen for the distance-dependent ground-seeking utility function: k and k_z. In trials shown in Section 5.1.2, values of $k = 0.02$ and $k_z \in [0.3, 0.5]$ exhibited good performance.

Lemma 5. *The distant-dependent ground-seeking utility function $U_{G,af}(d, z) = (1 - e^{-kd})\frac{1}{z+1}$ (and also $U_{G,af}(d, z) = (1 - e^{-kd})e^{-k_z z}$ or either function without distance dependence) satisfies the potential game formalism according to Definition 21 and Definition 22.*

Proof. This proof follows the same reasoning as shown in Proposition 1 and expresses the global potential Φ as follows (z_i is a shorthand notation for the z-component of action a_i).

$$\Phi(a_i, a_{-i}) = \left((1 - e^{-kd_i}) \frac{1}{z_i + 1} + \sum_{j \in \mathcal{P} \setminus \{i\}} (1 - e^{-kd_j}) \frac{1}{z_j + 1} \right)$$

The result then follows from the defining equation for potential games $U_i(a_i') - U_i(a_i) = \Phi(a_i', a_{-i}) - \Phi(a_i, a_{-i})$. □

Ground-averse utility Similar to ground-affinity, the ground-averse utility function relies on the z-coordinate of an action as input. In this case however, actions with a z-coordinate close to a desired target height are more highly rewarded. To ensure that this type of utility function has a unique maximum, a normal distribution-based formulation centered around a desired target height z_{des} is used. This utility function can potentially speed up the assembly of tall target structures and is formulated as follows.

$$U_{\text{G,av}}(z) = e^{\frac{-(z - z_{\text{des}})^2}{2\sigma^2}} \tag{25}$$

As before, one can include a distance-dependent term, which makes agents more ground averse the farther (or closer) they are located from the target configuration.

$$U_{\text{G,av}}(d, z) = \begin{cases} e^{\frac{-(z - z_{\text{des}})^2}{2\sigma^2}} (1 - e^{-kd}) & \text{higher influence farther from } \mathcal{C}_T \\ e^{\frac{-(z - z_{\text{des}})^2}{2\sigma^2}} e^{-kd} & \text{higher influence closer to } \mathcal{C}_T \end{cases} \tag{26}$$

Values of $\sigma \in [2, 10]$ exhibited good performance in simulation.

Lemma 6. *The ground-averse utility function $U_{G,av}(z) = e^{\frac{-(z - z_{des})^2}{2\sigma^2}}$ (with or without a distance-dependent term) satisfies the potential game formalism according to Definition 21 and Definition 22.*

Proof. This proof is equivalent to Lemma 5 with the only difference being the substitution of the exact utility term. Since the ground-averse utility function can be decomposed in the exact same way, the result still holds. □

Obstacle-averse utility The obstacle-averse utility function defines utility in terms of distance to obstacles and causes agents to avoid moving too close to static obstacles in the environment. In essence, this utility function acts like a potential field around obstacles and is formulated as follows.

$$U_{\mathrm{O,av}}(d_o) = u_{\min} e^{-k_o d_o} \tag{27}$$

In this formulation, $d_o = d_o(a_i) = \min_{o_j \in O} \|a_i - o_j\|_{L_2}$ where O is the set of obstacle positions in the environment. Unlike the aforementioned utility functions, obstacle-averse utility punishes proximity to obstacles in the sense that it assigns a negative utility value to an action. The parameters that need to be chosen here are a gain term k_o and a minimum utility value (or a maximum negative utility that is awarded to actions). Values of $k_o \in [1.0, 2.0]$ showed good performance in simulation. The value of $u_{\min}$ is required to be in the interval $u_{\min} \in [-1, 0)$ such that the obstacle-averse utility term remains in the negative unit interval $[-1, 0)$.

Lemma 7. *The obstacle-averse utility function $U_{O,av}(d_o) = u_{min} e^{-k_o d_o}$ satisfies the potential game formalism according to Definition 21 and Definition 22.*

Proof. This proof is equivalent to Lemma 5 with the only difference being the substitution of the exact utility term. Since the obstacle-averse utility function can be decomposed in the exact same way, the result still holds. □

5.2.2 Utility Functions as Weighted Sums

The above-mentioned utility function components can be combined as linear weighted sums and still satisfy the potential game requirement; and, thus, allow the computation of utility values with local information only. The following formulation combines the presented utility functions in Section 5.2.1 as such sums and enables the creation of scenario-specific behaviors of the overall system. For example, one could create a utility function that causes agents to "melt" any three-dimensional configuration into to a two-dimensional one by picking a set of coefficients $c = [c_1, c_2, c_3, c_4, c_5] = [0, 0, 1, 0, 0]$.

$$U = c_1 U_{\mathrm{T,af}} + c_2 U_{\mathrm{D,af}} + c_3 U_{\mathrm{G,af}} + c_4 U_{\mathrm{G,av}} + c_5 U_{\mathrm{O,av}} \tag{28}$$

Here, the utility is not limited to the unit interval but can take on values in the interval $U \in [-1.0, 3]$ for values $c_i \in [0, 1]$. This range for coefficients c_i is chosen such that individual utility component values remain in the unit interval.[3] The following lemma proves that this type of utility function still satisfies the potential game formalism.

Lemma 8. *Given any set of coefficients $c_i \in [0, 1]$, $\forall i \in \{1, 5\}$ the weighted sum utility function shown in Equation 28 satisfies the potential game formalism according to Definition 21 and Definition 22.*

Proof. This proof is based on Lemma 1, Lemma 4, Lemma 5, Lemma 6, and Lemma 7 and expresses an agent i's utility as follows.

$$U_i(a_i) = c_1 U_{i,\mathrm{T,af}}(a_i) + c_2 U_{i,\mathrm{D,af}}(a_i) + c_3 U_{i,\mathrm{G,af}}(a_i) + c_3 U_{i,\mathrm{G,av}}(a_i) + c_5 U_{i,\mathrm{O,av}}(a_i) \tag{29}$$

Since this formulation of the utility function still only depends on an agent's individual state and does not introduce coupling between agents, the decomposition of the global potential (as shown in Lemma 4) is still possible. Therefore, the defining equation $U_i(a_i') - U_i(a_i) = \Phi(a_i', a_{-i}) - \Phi(a_i, a_{-i})$ also holds and the weighted sum utility function satisfies the potential game formalism. □

Note that while any set of coefficients $c_i \in [0, 1]$ yields a utility function that conforms to the potential game formalism, not every set of coefficients will lead to convergence to the target configuration under Theorem 9 and Theorem 10. Theorem 10 explicitly states convergence to the potential function maximizer. Therefore, only those utility functions guarantee convergence whose maximum is located at an action $a_i \in \mathcal{C}_T$ (as was shown in Section 4.2). For example, the set of coefficients $c = [c_1, c_2, c_3, c_4, c_5] = [1, 0, 0, 0, a]$ with $a \in [0, 1]$, $k_o = 1.0$, and $u_{\min} \in [-1.0, 0)$ satisfies this criterion and adds obstacle avoidance behavior to the target-seeking utility function.

Theorem 11. *Given a weighted sum utility function U (Eqn. 28) with any set of constant coefficients c_i such that $c_i \in [0, 1]$, $\forall i \in \{1, 5\}$, the self-reconfiguration algorithm (Algorithm*

[3] Note that the restriction to coefficient values $c_i \in [0, 1]$ is an arbitrary choice to enable more predictable utility values and more consistent convergence times. However, the results in this section and the following still hold for any finite real values of c_i.

4 or Algorithm 5) is still guaranteed to converge to the target configuration under Theorem 9 and Theorem 10 as long as the maximum of U is achieved at target positions $a_j \in \mathcal{C}_T$.

Proof. The requirement that the utility function maximizer is located at a target position $a_j \in \mathcal{C}_T$ ensures that the potential function maximizer is located at the target configuration $\mathcal{C}_T$. As such, constant coefficients c_i induce an irreducible time-homogeneous Markov process for which Theorem 9 and Theorem 10 guarantee convergence. Irreducibility is given by the fact that the transition probability of any possible action (according to Theorem 9)

$$p_{ij} = \begin{cases} q_{ji} e^{\frac{1}{\tau}(U_k(a'_k) - U_k(a_k))} & \text{if } e^{\frac{1}{\tau}(U_k(a'_k) - U(a_k))} \frac{q_{ji}}{q_{ij}} \leq 1 \\ q_{ij} & \text{o.w.} \end{cases}$$

are guaranteed to be nonzero irrespective of the utility values.[4] □

5.2.3 Utility Function Switching

This section introduces a utility switching scheme that enables agents to adjust their utility function during runtime and independently of other agents in the system. Switching between utility functions can be thought of as a modification of the coefficients in the weighted sum utility function in Eqn. 28 or a complete replacement of the utility function altogether. As stated in Section 5.2.1 however, any utility function that is used in this switching scheme is required to depend only on the individual agent's state or the state of static objects in the environment (to avoid additional coupling between agents). *State-based switching conditions* are then introduced that allow individual agents to determine autonomously when to switch to another utility functions. These switching conditions depend only on an agent's individual state (such as the height above ground) or quantities that can be derived from it without communicating or cooperating with other agents (such as the distance to the target configuration or the distance to a static obstacle in the environment).

The utility switching scheme is designed as a finite state machine (FSM) in which each state represents a specific utility function to be used by an agent (an example is shown in Fig.

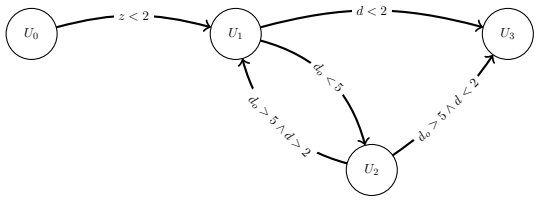
[4] Even for the case when the utility $U_i(a_i, a_{-i}) = 0$, $\forall a_i \in \mathcal{E}$, the performance would deteriorate to naive random walk, but the convergence guarantee would still hold. In fact, any finite utility value results in nonzero transition probabilities.

29b). Agents are initialized in a state s_0 and are only allowed to transition between states as specified by a transition function δ. This transition function encodes a user-specified transition graph G_{tr} that contains utility modes as well as switching conditions between modes. Each switching condition represents an edge between two states (e.g. $\delta(s_0, x)$ represents a switching condition between states s_0 and s_1 in Fig. 29b). Note that an edge does not have to exist between every pair of states. However, the input transition graph G_{tr} is required to be a directed graph that is weakly connected. Moreover, G_{tr} has to contain a unique global sink, i.e. a unique absorbing state s_M of the state machine that cannot be left once reached.[5] In other words, agents stop switching utility modes as soon as they reach the final state s_M (in Fig. 29b $s_M = s_3$). The utility function associated with the final state s_M is required to be target-seeking (or in general reach a maximum at target positions $a_i \in \mathcal{C}_T$) such that convergence to the target configuration can be guaranteed (see Theorem 12). The utility functions of all other state of the FSM, however, do not have to be target-seeking and are therefore not required to reach their maximum value at target positions. For example, a utility function that causes the configuration to "melt" could have a maximum at positions $z = 1$ on the ground plane.

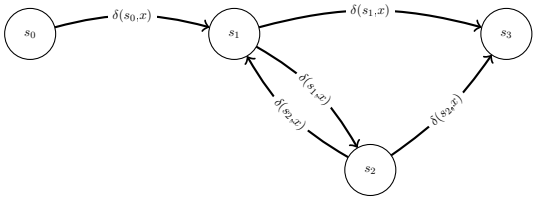
In this section, as a general guideline, utility functions are used that reach their maximum value at the respective switching condition to the next state. For example, U_0's global maximum is located at a state that triggers the switching condition $z < 2$ in Fig. 29a. However, note that in finite spaces this is not required as it can be guaranteed that agents reach *state-based switching conditions* even without any incentivization (using only naive random walk). The specific design of switching conditions and utility functions in this section however is intended to speed up the convergence to these switching conditions. The following definition makes the notion of switching conditions more concrete.

Definition 31 (State-based Switching Condition)**.** *A state-based switching condition is one that can be decided upon by an individual agent without the cooperation or information from other agents. It is triggered by a change in an agent's individual state (i.e. its position a_i in*

[5] A sink node v_i in a graph is a node with outdegree 0, i.e. no path exists from v_i to any other node in the directed graph.



(a) Example of a user-specified transition graph G_{tr} which contains utility functions as its nodes and switching conditions as its directed edges. This transition graph is also used in Example 1.



(b) Example of a finite state machine showing the set of states $S = \{s_0, s_1, s_2, s_3\}$ and the transition function δ. The final state s_3 is the only absorbing state of the FSM.

Figure 29: Example of a user-specified transition graph and the utility switching state machine constructed from it.

the environment $\mathcal{E}$), a component thereof, or a quantity that can be derived from the agent's state based on its own knowledge about $\mathcal{E}$. As such, the following switching conditions are considered.

- *Target distance-based: $d \gtrless d_{des}$ (with $d_{des} \geq 0$)*
- *Height-based: $z < z_{des}$ (with $z_{des} \geq 1$)*
- *Obstacle distance-based: $d_o \gtrless d_{o,des}$ (with $d_{o,des} \geq 0$)*[6]

Note that conditions such as $z > z_{\text{des}}$ or $|\mathcal{N}_i| \gtrless k$ (the cardinality of an agent's neighborhood set) are explicitly excluded from the set of potential switching conditions because triggering these conditions requires either the cooperation (for reaching a certain target height $z > z_{\text{des}}$) or information (for computing the cardinality $|\mathcal{N}_i|$ of an agent's neighborhood) from other agents. As such, coordination between agents would be required to trigger these conditions. However, since every agent acts in a purely self-interested way, agents are not incentivized to cooperate with other agents. The definition of switching conditions allows the automatic design of utility functions for the individual modes of the state machine by employing the formulations from Section 5.2.1. A height-based switching condition of the form $z < z_{\text{des}}$, for example, can use ground-seeking utility functions of the form $U_{\text{G,af}} = e^{-k_z z}$ (as shown in Eqn. 23). The following definition incorporates switching conditions from Def. 31 to rigorously specify the finite state machine used for utility switching.

[6] *Clearly, an obstacle-based switching condition is only useful if there exist obstacles in the environment. Otherwise, by default $d_o = 0$.*

Definition 32 (Utility Switching State Machine)**.** *Let the utility switching state machine USSM be a quintuple* $(\Sigma, S, s_0, \delta, F)$ *representing a deterministic finite state machine with the following components (for example [27]).*

- Σ *is an input alphabet (or a finite nonempty set of inputs). Specifically,* Σ *is the set of all integer triplets* $(d_i, z_i, d_{o,i})$ *that are possible for agent* i *within a given finite environment* $\mathcal{E}$*. These variables represent an agent's distance* d_i *to the target configuration, an agent's height above ground* z_i*, and an agent's distance* $d_{o,i}$ *to the nearest obstacle.*[7]

- S *is a finite, nonempty set of states, where each state represents a utility function to be used by agents (i.e. a node in the transition graph). Specifically,* $S = \{s_0, s_1, \ldots, s_M\}$ *where* M *is the cardinality of the set* S *or the number of utility modes specified in the transition graph (see Fig. 29a).*

- $s_0 \in S$ *is the initial state or the initial utility mode.*

- δ *is the state transition function* $\delta : S \times \Sigma \to S$ *that maps a current state based on inputs* $(d_i, z_i, d_{o,i})$ *to new states in* S*. The transition function encodes the user-specified transition graph* G_{tr} *(see Fig. 29b). This graph* G_{tr} *has to be weakly connected such that a directed path exists from any state* s_i *to the final state* s_M*. More specifically, the state* s_M *has to be the unique global sink in* G_{tr}*. Note that* δ *is a total function, i.e. it is defined for every possible combination of state* s_i *and input* $x = (d_i, z_i, d_{o,i})$ *such that every combination results in a deterministic output.*

- F *is the set of final states and contains a single element such that* $F = \{s_M\}$ *(the final utility mode using a target-seeking utility function). In other words the state* s_M *is the only absorbing state of the state machine.*

A single utility switching state machine USSM is defined for the aggregate system. However, every individual agent i, instantiates and executes its own copy of that state machine and determines transitions in that state machine based on its own state-based input

[7] *Note that the distance to the target configuration* d_i *and the distance to obstacles* $d_{o,i}$ *are real values and thus have to be rounded to integers. Otherwise the alphabet* Σ *would contain infinitely many elements which violates the definition of a finite state machine.*

$(d_i, z_i, d_{o,i})$. As such, all agents do not have to execute the same utility mode at the same time. The following example clarifies the construction of the utility switching state machine based on a user-specified transition graph which contains utility modes and switching conditions defined in Def. 31.

Example 1. Consider the transition graph in Fig. 29a. This graph contains four utility modes $\{U_0, U_1, U_2, U_3\}$ that will be mapped to a set of states $S = \{s_0, s_1, s_2, s_3\}$ and five switching conditions that will be encoded in δ. In this example, these utility functions U_i are defined as follows.

$$\begin{aligned} U_0 &= e^{-0.2z} \\ U_1 &= e^{-0.1d} + (1 - e^{-0.02d})e^{-0.2z} \\ U_2 &= e^{-0.1d} + (1 - e^{-0.02d})e^{-0.2z} + u_{\min}e^{-k_o d_o} \\ U_3 &= e^{-0.1d} \end{aligned}$$

Utility function U_0 is a purely ground-seeking utility function and thus "melts" the configuration into a two-dimensional configuration. U_1 combines a target-seeking term with a distance-dependent ground-seeking term that decreases in influence as agent i moves closer to $\mathcal{C}_T$. U_2 adds obstacle avoidance behavior to utility function U_1 in case an agent moves too close to obstacles. Finally, U_3 is a purely target-seeking utility function and reaches its maximum at target positions $a_i \in \mathcal{C}_T$ as required by Def. 32.

Having defined the set of states, we can set the initial state to s_0 and determine the set of final states $F = \{s_3\}$. Most importantly, we can now define the transition function and

ensure that δ is a total function.

$$
\begin{aligned}
\delta &= \{\delta(s_0,x), \delta(s_1,x), \delta(s_2,x), \delta(s_3,x)\} \\
\delta(s_0,x) &= \begin{cases} s_1 & \text{if } z(x) < 2 \\ s_0 & \text{o.w.} \end{cases} \\
\delta(s_1,x) &= \begin{cases} s_3 & \text{if } d(x) < 2 \\ s_2 & \text{if } d_o(x) < 5 \wedge d(x) > 2 \\ s_1 & \text{o.w.} \end{cases} \\
\delta(s_2,x) &= \begin{cases} s_3 & \text{if } d_o(x) > 5 \wedge d(x) < 2 \\ s_1 & \text{if } d_o(x) > 5 \wedge d(x) > 2 \\ s_2 & \text{o.w.} \end{cases} \\
\delta(s_3,x) &= s_3
\end{aligned}
$$

Using the alphabet Σ as defined in Def. 32, the utility switching state machine is now fully defined (see Fig. 29b) and can be executed by each agent.

Based on the outlined utility switching setup, the following theorem proves convergence to the desired target configuration $\mathcal{C}_T$.

Theorem 12. *Given a transition graph G_{tr} that fully defines a utility switching state machine (USSM, Def. 32), the aggregate self-reconfigurable system converges to the unique stochastically stable state $\mathcal{C}_T$ if the utility function U_M of the final state $s_M \in S$ is target-seeking.*

Proof. This result is shown in two steps. First, it is demonstrated that an individual agent is guaranteed to reach the final state s_M of the USSM. Second, we show that the results from Theorem 9 (or Corollary 2) and Theorem 10 are applicable to prove stochastic stability of $\mathcal{C}_T$.

Given a finite environment and transition probabilities that are non-zero for every possible action of every agent regardless of the utility function used (according to Theorem 9 or Corollary 2), any possible action path of an individual agent has a non-zero probability

of occurring. Since this is true irrespective of the utility function used (as shown in Theorem 11), it is also true for any sequence of utility functions used, i.e. for any number of switches between states in the USSM. As such, an action path leading a single agent to a position that satisfies a switching condition to the final state s_M has non-zero probability and will eventually occur. This is true for every agent i in the aggregate system, which means, eventually all agents will switch to state s_M.

Once all agents have switched to state s_M with utility function U_M, they will use a fixed utility function and the requirements for Theorem 9 are satisfied. Since U_M is target-seeking, it reaches its maximum at target positions $a_i \in \mathcal{C}_T$. Therefore, the maximum global potential is reached in a state in which every agent is at a target position $a_j \in \mathcal{C}_T$. Theorem 9 (or Corollary 2) can then be applied to show convergence to the desired target distribution and Theorem 10 guarantees stochastic stability of the target configuration $\mathcal{C}_T$. □

5.2.4 Results

In this section we compare the performance of four variations of the decentralized implementation of Algorithm 5 which use different incentivization schemes. The basic target-seeking utility shown in Def. 22 and Eqn. 21 is compared against an adaptive target-seeking utility (using results from Section 5.1), a utility function switching scheme (according to Section 5.2), and a combination of utility function switching and an adaptive learning rate. As before, both algorithm were implemented and evaluated in Matlab. The adaptive learning rate strategies used the same parameters as shown in Section 5.1.2 where $\tau_{nom} = 0.01$, $\tau_{max} = 1.0$, and $N = 15$. The utility function switching scheme used three utility functions and two switching conditions summarized as follows (note that d is the distance of an action to the closest target position and z is an action's height above the ground plane).

$$\begin{aligned} U_1 &= U_{\max} e^{-0.2z} \\ U_2 &= U_{\max} e^{-0.1d} + (1 - e^{-0.02d}) e^{-0.2z} \\ U_3 &= U_{\max} e^{-0.1d} \end{aligned} \tag{30}$$

Utility function U_1 is a purely ground-seeking utility function and thus "melts" the configuration into a two-dimensional configuration. An agent i switches to utility function U_2

when the first trigger condition $z \leq 2$ is met, i.e. when agent i occupies a position on the ground plane. U_2 combines a target-seeking term with a distance-dependent ground-seeking term that decreases in influence as agent i moves closer to $\mathcal{C}_T$. Once the second switching condition $d \leq 4$ is triggered, agent i switches to a purely target-seeking utility function U_3.

As in Section 5.1.2, agents compute their actions according to Section 4.2.2. Agents' actions were restricted by other agents' actions and the ground plane. No obstacles were added to the environment in this section. Three scenarios are presented below that highlight the strengths and weaknesses of all four incentivization schemes. For all scenarios below, ten trials were run for each incentivization scheme and for each scenario.

Low box configurations containing 36 agents without obstacles Fig. 30 shows the result of reconfiguration sequences from an initial box configuration to a target box configuration. These boxes had dimensions $3 \times 3 \times 4$ cubes (for a total of 36 cubes) and were spaced 8 grid cells apart (the specific setup is shown in the inset of Fig. 30). Convergence is achieved when the global potential reaches a value of $\Phi = 36$ (represented by the horizontal red line in the figure). The vertical lines in Fig. 30 represent the average time to convergence for the four different approaches. The figure shows similar convergence times for all four approaches with a small speedup for the utility switching approaches. This is an expected result for two reasons. First, the initial and target configuration are located too close to each other for dendrite-like sub-structures to form (as shown in Fig. 31) because agents can essentially form bridges from the initial to the target configuration. Second, the initial configuration is not tall enough for the "melting" utility function U_1 to have a significant effect on the speedup of the reconfiguration sequence.

Tall box configurations containing 48 agents without obstacles Fig. 32 shows the result of reconfiguration sequences from an initial box configuration to a target box configuration. These boxes had dimensions $2 \times 3 \times 8$ cubes (for a total of 48 cubes) and were spaced 15 grid cells apart (the specific setup is shown in the inset of Fig. 32). Convergence is achieved when the global potential reaches a value of $\Phi = 48$ (represented by the horizontal red line in the figure). The vertical lines in Fig. 32 represent the average time to convergence

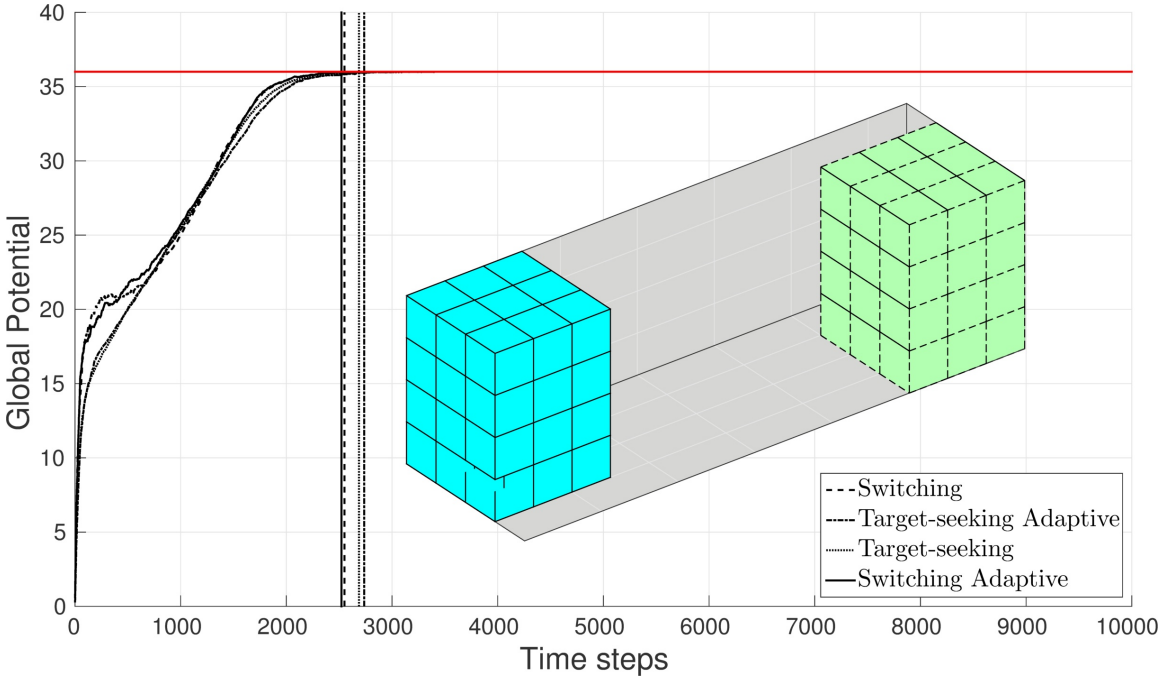


Figure 30: Convergence times for a configuration containing 36 agents using four different incentivization strategies: a fixed target-seeking utility (Section 4.2), an adaptive target-seeking utility (Section 5.1), switching utility functions (Section 5.2), as well as adaptive switching utility functions (a combination of Section 5.1 and Section 5.2). The initial and target box configurations are spaced eight units apart.

for the four different approaches. As can be seen in the figure, the adaptive utility switching approach converges fastest on average, closely followed by the basic utility switching approach. Both target-seeking (fixed and adaptive) approaches showed significantly slower convergence with the adaptive target-seeking method faring notably better. The reason for this discrepancy is that the target-seeking utility function encourages the formation of dendrite-like substructures as shown in Fig. 31 which are unlikely to be resolved within the given time horizon of 10,000 time steps. Note that failed trials also mean that the average global potential for certain approaches does not reach the maximum possible potential. This is shown for the target-seeking, adaptive target-seeking, and basic switching utility traces in Fig. 32 that do not reach $\Phi = 48$ after 10,000 time steps.

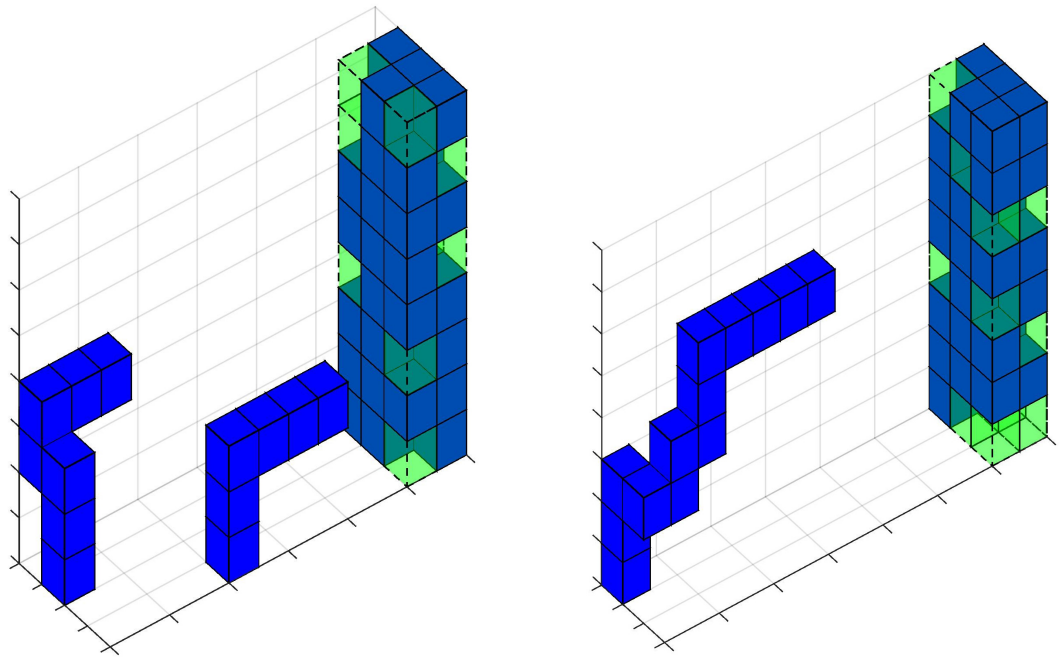


Figure 31: Examples of dendrite-like sub-configurations forming that prevent the assembly of the target configuration within the 10,000 time step horizon.

Exogenous versus endogenous switching This experiment presents a twist on the switching paradigm where two switching strategies are compared. The first strategy uses the previously outlined endogenous switching where agents decide autonomously based on their own state when to switch. The second strategy uses exogenous switching, where all agents

receive a switching signal in a broadcast fashion and switch at the same time. The latter can be thought of as an external entity initiating a switch based on observed information about the aggregate system. The goal of this experiment is to compare the performance of these switching approaches and evaluate whether exogenous switching, which has access to global information, results in faster convergence times. The exogenous switch is based on the observed global potential (i.e. cumulative utility of all agents in the system). Three switching thresholds are compared: 80%, 90%, 95% of the maximum achievable global potential for a given utility mode. For example, a switching threshold of 95% using a ground-seeking utility function "melts" an initial three-dimensional configuration almost completely into a two-dimensional configuration before switching to the next utility mode. A threshold of 80% switches while sub-configuration still exhibits three-dimensional structure.

Similar to the previous experiment, here, reconfiguration sequences from an initial box configuration to a target box configuration are shown. These boxes had dimensions $2 \times 3 \times 8$ cubes (for a total of 48 cubes) and were spaced 15 grid cells apart (the specific setup is shown

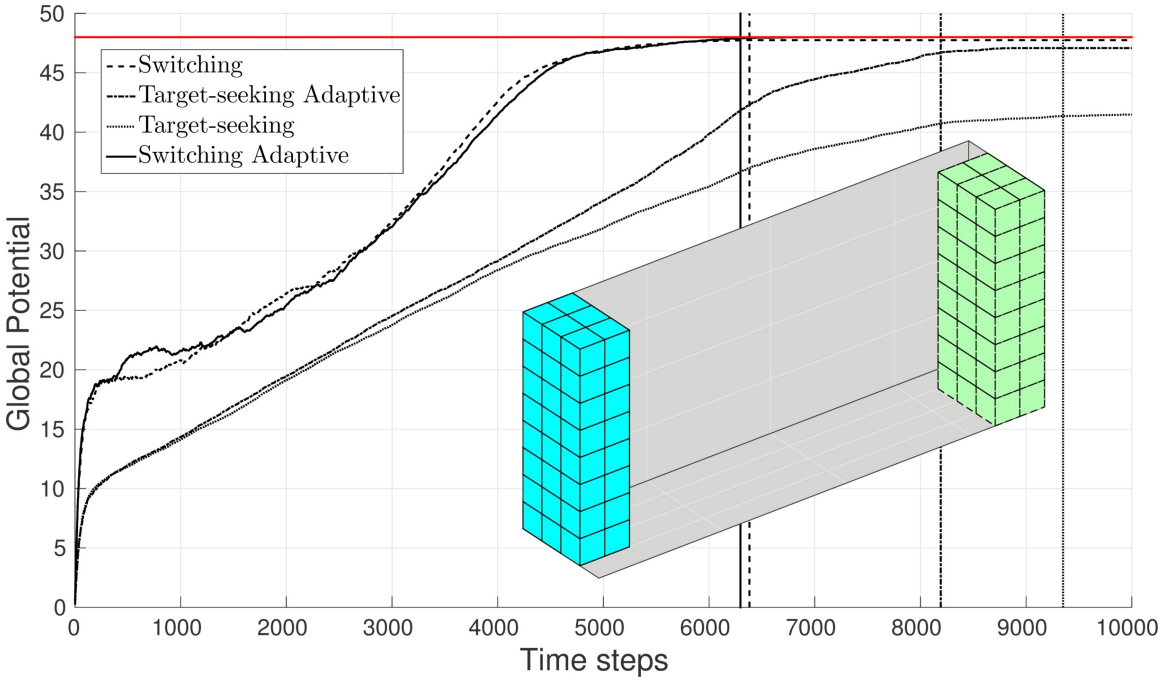


Figure 32: Convergence times for a configuration containing 48 agents using four different incentivization strategies: a fixed target-seeking utility (Section 4.2), an adaptive target-seeking utility (Section 5.1), switching utility functions (Section 5.2), as well as adaptive switching utility functions (a combination of Section 5.1 and Section 5.2).

in the inset of Fig. 33). Again, convergence is achieved when the global potential reaches a value of $\Phi = 48$ (represented by the horizontal red line in the figure). The vertical lines in Fig. 33 represent the average times to convergence for the four different approaches. The basic endogenous switching strategy outlined in Section 5.2 performs the best with almost a 10% speedup over the closest exogenous switching approach that uses a threshold of 95%. The main reason for the slower convergence of exogenous switching was its comparable crudeness. Whereas endogenous switching allows fine-grained control over when individual agents switch, exogenous switching treats the configuration as an aggregate system and switches based on the global state. Therefore, if the switching threshold was set too low, the configuration tended to switch too early and dendrites formed as shown in Fig. 31. If the switching threshold was set too high, the configuration was almost completely "melted" into a two-dimensional structure, which slowed down convergence and caused agents to spread out farther as shown in Fig. 34.

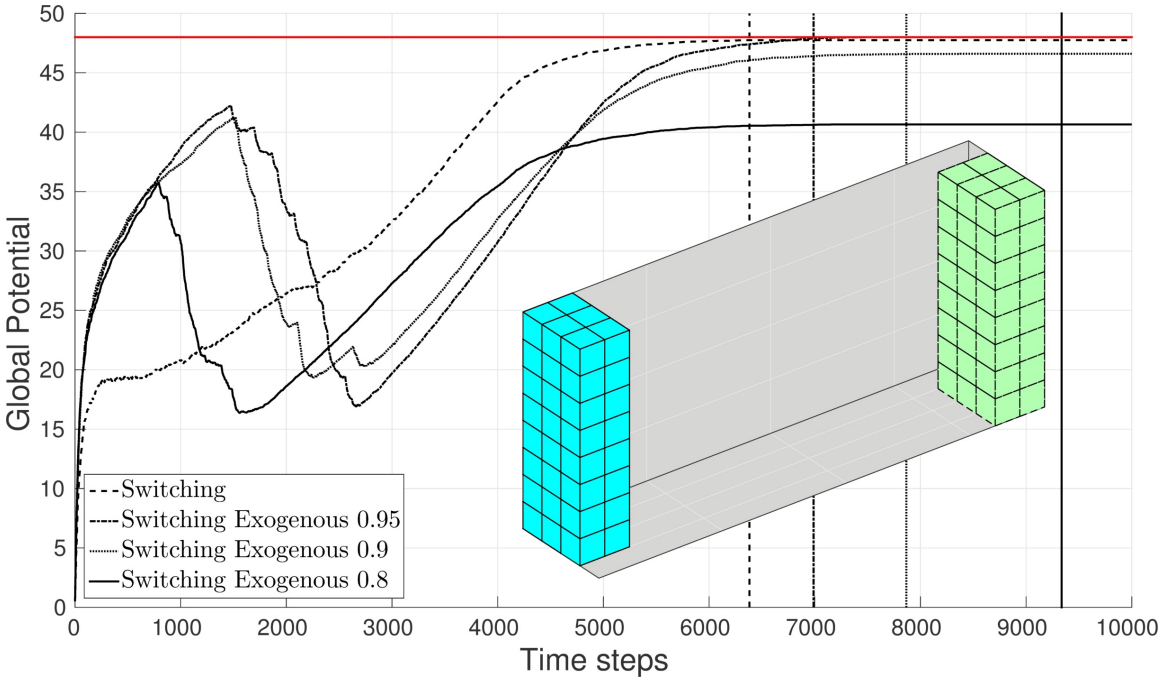


Figure 33: Convergence time results for an example comparing endogenous and exogenous switching strategies. The basic switching strategy of Section 5.2.3 is compared against three exogenous switching strategies using different switching thresholds: 80%, 90%, 95% of the maximum achievable global potential for a given utility mode.

Note that the reason, the global potential traces shown in Fig. 33 do not reach 80%,

90%, or 95% of the maximum achievable global potential before switching is that the shown results are averaged over ten trials. Switches for the individual trials still occurred at these thresholds, however, the averaged utility time series cannot visualize that fact. Also note that the drop in global potential for the exogenous switching strategies is caused by a sudden change in utility function of all agents simultaneously. While these drops are also noticeable in the endogenous switching strategy, they are less pronounced.

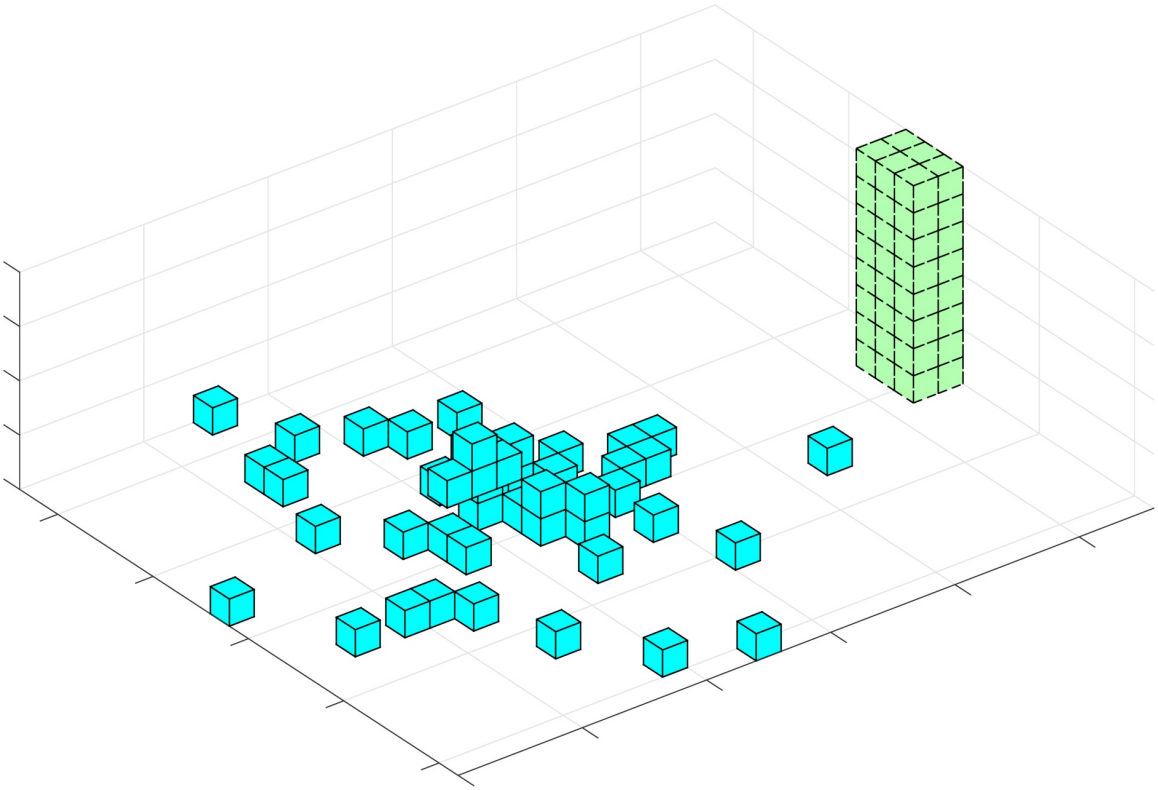


Figure 34: Examples of a spread out configuration caused by a too large switching threshold in an exogenous switching example.

5.3 Conclusions

This chapter has introduced two extensions to the basic game-theoretic algorithm shown in Section 4.2 that allow the runtime adjustment of the learning rate of Algorithm 5 on the one hand and the switching of utility functions on the other hand. Various utility functions have been discussed in the context of switching together with desirable utility function properties that allow the decentralized execution and avoid additional coupling between agents. Additionally, a systematic method for designing switching sequences and switching

conditions has been introduced.

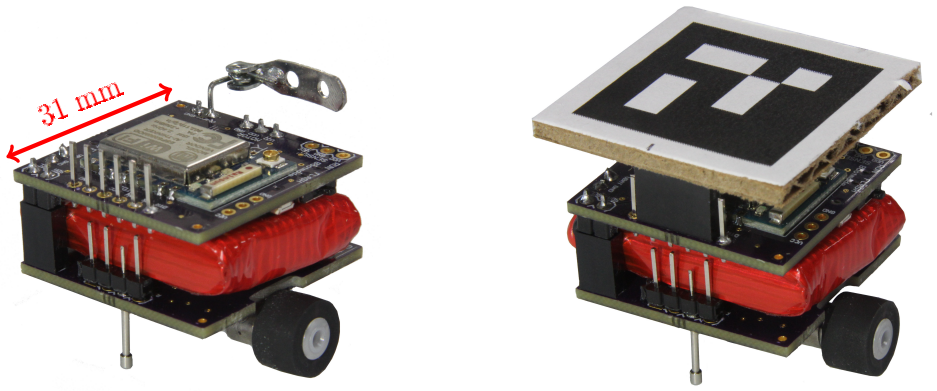
The theoretic contributions in this chapter show that convergence guarantees are retained for both adaptive methods while simulation results (shown in Section 5.1.2 and Section 5.2.4) indicate that improved convergence rates for both approaches can indeed be achieved for certain scenarios. This is the case particularly for scenarios which contain tall initial and/or target configurations that are spaced far apart. Scenarios containing obstacles were solved faster using the adaptive learning rate (with or without utility function switching). No significant improvement was observed for trivial scenarios with configurations with small numbers of agents or initial and target configurations that were located close to each other.

The next chapter will focus on showing how the algorithms presented in Chapter 4 and 5 can be instantiated on robotic hardware. Specifically, Chapter 6 will explain in detail the development of a multi-robotic testbed including the design of a novel miniature robot, the GRITSBot.

Chapter VI

ROBOTIC IMPLEMENTATION

This chapter discusses the design and development of a novel miniature wheeled robot called the GRITSBot, a remotely accessible swarm-robotic testbed called the Robotarium, and the implementation of decentralized game-theoretic self-reconfiguration on a team of GRITSBot robots (see Fig. 35). While a number of wheeled robots are available off the shelf, the GRITSBot and the Robotarium serve a larger purpose than to create another robot and to simply instantiate a self-reconfiguration algorithm. The overarching goal of the Robotarium is to create accessible multi-robotic hardware and lower the barrier to entrance into the field of multi-agent robotics. Therefore, this chapter does not only elaborate on the instantiation of the game-theoretic self-reconfiguration algorithm (Algorithm 5) on the GRITSBots but also illuminates the concept of the Robotarium and its mission of democratizing multi-agent robotics.



(a) Assembled robot with motor and main layer.

(b) Assembled robot with tracking tag.

Figure 35: Isometric and top view of the GRITSBot.

Before introducing the Robotarium in detail, however, the next section discusses how the self-reconfiguration algorithm can be executed on a team of robots. Therefore, Section

6.1 presents the dynamical model of the GRITSBot, derives low-level velocity and position control laws, and finally shows how Algorithm 5 can be mapped onto the continuous dynamics of the robot. Experimental results with eight robots then demonstrate the feasibility of two-dimensional self-reconfiguration in practice. The remainder of this chapter is dedicated to the presentation of the Robotarium. Specifically, in Section 6.2 we introduce a high-level view of the concept and the objectives of the Robotarium. Section 6.3 then outlines how the Robotarium is structured and, in particular, how the explicit focus on being a remote-access research platform informs the design of the testbed itself and the robots it contains. The specifications and detailed designs of both the GRITSBot robots and the full Robotarium testbed are then described in Section 6.4 and 6.5.

6.1 Self-Reconfiguration on Robots

The self-reconfiguration algorithms presented in Chapter 3 to 5 implicitly rely on the discrete motion primitives imposed by the sliding cube model in Section 2.3. Whereas the sliding cube model can be simulated in a straightforward fashion, the execution on robots requires the mapping of this motion model onto the specific continuous dynamics of the chosen hardware platform. In this section, we therefore describe the instantiation of the game-theoretic self-reconfiguration algorithm (Algorithm 5) on the GRITSBots (see Fig. 35). Specifically, this section presents low-level controllers for velocity and position control of the robots and elaborates on the mapping of discrete actions to the continuous dynamics of the robot. We first outline the dynamical model of the robot before discussing the overall system architecture that allows the integration of high and low-level control as well as the general interaction with the Robotarium.

For the purpose of self-reconfiguration on robots, this section just briefly introduces the GRITSBot while an in-depth presentation of the design and specification details is deferred until Section 6.4. The GRITSBot is a novel differential-drive robot with a modular layered design as shown in Fig. 35. While similar in capabilities to most commonly used platforms for multi-agent research, the GRITSBot features a significantly smaller footprint (approximately 30×30 mm) and lower cost (approximately \$40 in parts). It is equipped with

an accurate locomotion system, infrared-based distance sensing, and WiFi communication that allows the GRITSBot to be easily integrated into an existing network. Additionally the robot is equipped with convenience features that allow a human operator to effortlessly control a large collective of robots. These features include automatic battery charging, wireless reprogramming, and automatic sensor calibration and will be discussed in Section 6.5.

6.1.1 Low-level Control

Dynamics The GRITSBot is a differential drive robot whose state can be described by a three-dimensional state vector containing its position in the plane (p_x, p_y) and its heading angle θ with respect to a global coordinate system. Its state vector can thus be summarized as $x = [p_x, p_y, \theta]$. A common dynamical model for differential drive robots is the unicycle model, which captures the nonlinear dynamics of the robot and its nonholonomic constraints, more specifically, its non-integrable velocity constraints. Intuitively these constraints describe the fact that a unicycle-type robot cannot move perpendicular to the direction of its wheels (see [45]), i.e. instantaneously move sideways. These dynamics can be described using the following equation.

$$\dot{x} = f(x,u) = \begin{bmatrix} v\,cos(\theta) \\ v\,sin(\theta) \\ \omega \end{bmatrix} \tag{31}$$

Note that the linear and angular velocity v and ω appear in those dynamics equations as inputs to the system. Used for convenience and a cleaner formulation, these velocity inputs need to be mapped to wheel velocities that directly control the motors of the robot. The mapping to wheel velocities is given by the following one-to-one mapping.

$$\begin{bmatrix} w_L \\ w_R \end{bmatrix} = \begin{bmatrix} (2v - \omega K_B)/K_R \\ (2v + \omega K_B)/K_R \end{bmatrix} \tag{32}$$

Note that K_R is the wheel diameter and K_B is the wheel base shown in Figure 36. The parameter values are summarized in Table 7. Additionally, since the GRITSBot's motors are stepper motors, these wheel velocities need to be mapped to delays between individual

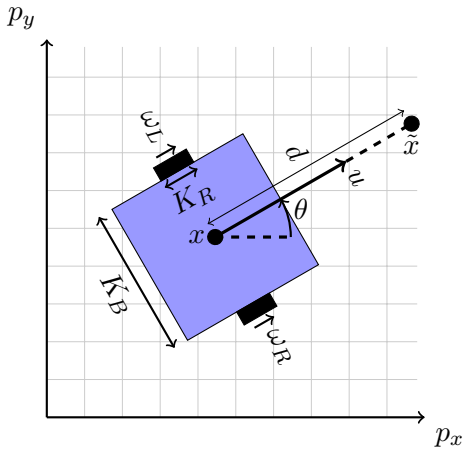


Figure 36: Unicycle model for differential drive robots based on the geometry of the GRITSBot.

Table 7: Parameters used for the control of the GRITSBot.

Parameter	**Value**
K_B	31 mm
K_R	10 mm
d	5 mm

motor steps. The mapping from linear and angular velocities to wheel velocities to delays between steps is computed directly on the motor board of the GRITSBot (see Fig. 41a and 42a) based on the fact that the GRITSBot's motors are capable of 40 steps per rotation. A number of approaches are available in the literature for controlling robots with unicycle dynamics (for example [45]). In the following sections we will show a feedback linearization-based position controller and a controlled Lyapunov function (CLF)-based pose controller.

Feedback Linearization-based Control One way of dealing with the non-holonomic nature of unicycle dynamics is to control a point located in front of the robot by some distance d instead of the true robot state located at the center of the wheel base (see Figure 36). Through this offset d one essentially creates a new state $\tilde{x}$ that is controlled instead of the original state x. Intuitively, controlling $\tilde{x}$ can be thought of as dragging the robot through an imagined rigid rod connected to the wheel base of the robot. The end of this rod is where this new state $\tilde{x}$ is located, which can be moved instantaneously in every direction

in the plane and is therefore holonomic (because no velocity constraints are associated with $\tilde{x}$). This state $\tilde{x}$ is diffeomorphic to the true state x and as such can be mapped directly to motion in the true state. The following mapping that is parametrized by the robot's orientation θ and the length d formalizes the mapping from $\tilde{x}$ to x.

$$\tilde{x} = x + \begin{bmatrix} d\,cos(\theta) \\ d\,sin(\theta) \\ 0 \end{bmatrix} \tag{33}$$

The dynamics of $\tilde{x}$ are then given by the following time derivative.

$$\dot{\tilde{x}} = \dot{x} + \begin{bmatrix} -d\,sin(\theta)\,\dot{\theta} \\ d\,cos(\theta)\,\dot{\theta} \\ 0 \end{bmatrix} = \begin{bmatrix} v\,cos(\theta) - d\,\omega\,sin(\theta) \\ v\,sin(\theta) + d\,\omega\,cos(\theta) \\ \omega \end{bmatrix} \tag{34}$$

The state $\tilde{x}$ can be controlled through feedback linearization if we limit ourselves to position control instead of full state control including orientation (this restriction is required to ensure that the matrix G shown below is invertible). Using a two-dimensional state space with $\tilde{x} = [p_x, p_y]$, we can formulate the velocity control law in the following compact form.

$$\dot{\tilde{x}} = \begin{bmatrix} \dot{p_x} \\ \dot{p_y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & d \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} = R(\theta)S(d) \begin{bmatrix} v \\ \omega \end{bmatrix} = G(\theta, d) \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{35}$$

Using the rotation matrix $R(\theta)$ and the scaling matrix $S(d)$ we can define the matrix $G(\theta, d) = R(\theta)S(d)$ and use G to define a mapping between the linear velocities of state $\tilde{x}$ and the linear and angular velocities of the original state as follows.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = G^{-1}(\theta, d) \begin{bmatrix} v_x \\ v_y \end{bmatrix} \tag{36}$$

Note that v_x and v_y are the linear velocities applied to the diffeomorphic state $\tilde{x}$ in the x and y direction. This the two-dimensional state $\tilde{x}$ is completely controllable in the plane and its dynamics are holonomic. The dynamics $\dot{\tilde{x}}$ are then feedback linearized as follows.

$$\dot{\tilde{x}} = R(\theta)S(d) \begin{bmatrix} v \\ \omega \end{bmatrix} = R(\theta)S(d)G^{-1} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = R(\theta)S(d)S(d)^{-1}R(\theta)^{-1} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \tag{37}$$

This transformed system now has linear dynamics (single integrator dynamics), where the input $v_{lin} = [v_x, v_y]$ can be directly used to control the diffeomorphic state $\tilde{x}$. The following expressions summarize how the actual control inputs v and ω are computed on the robot itself based on linear velocities v_{lin}.

$$\begin{aligned} v &= \cos(-\theta)v_x - \sin(-\theta)v_y \\ \omega &= \frac{1}{d}\left(\sin(-\theta)v_x + \cos(-\theta)v_y\right) \end{aligned} \tag{38}$$

The linear input velocities v_x and v_y in the above equation are computed using linear feedback position control. The position controller shown below derives linear velocities as inputs to the diffeomorphic state $\tilde{x}$ (i.e. v_x and v_y) based on the position error $e = \tilde{x} - x_D$ (where x_D is a desired target position).

$$v_{lin} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} = -Ke \tag{39}$$

Here, K is a gain matrix that is constant in this implementation but can alternatively be computed as the LQR gain matrix by solving an LQR problem at every time instant. This feedback law reduces the dynamics of the system to the following form.

$$\dot{\tilde{x}} = v_{lin} = -Ke \tag{40}$$

A Controlled Lyapunov Function Approach In this section we outline an approach for pose control (as opposed to the position control approach presented above) that is partially based on controlled Lyapunov functions. The method shown in this section leads directly to a smooth control law that allows position and orientation control of robots using unicycle dynamics. The derived control law is similar to the one shown in [2]. But instead of enabling steering, path-following, and navigation as shown in this paper, we simplify the control law to only enable a go-to-goal behavior. The desired target position being constant and time-invariant simplifies the design of the position control law as compared to [2]. The error in heading however is time-varying and as such allows no such simplification. Instead of a full CLF-based method, a proportional controller is therefore used here to control the heading of the robot. Using a target pose $r = [r_x, r_y, r_\theta]$ we can represent the position and orientation

error as follows. Note that the heading of the robot is not controlled to align with the target orientation r_θ but with the error vector $e = [e_x, e_y]^T$ instead for smoother motion.

$$\begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \\ atan2(e_y, e_x) \end{bmatrix} - \begin{bmatrix} p_x \\ p_y \\ \theta \end{bmatrix} \tag{41}$$

Using these error quantities, the full pose control law can be formulated that computes linear and angular velocities updates. We forgo the derivation of the CLF-based position control law and just note that the Lyapunov function $V(x) = \frac{e_x^2}{2} + \frac{e_y^2}{2}$ is used to derive it.

$$v = K_v \left(e_x cos(\theta) + e_y sin(\theta)\right) \tag{42}$$

$$\omega = \begin{cases} K_\omega \left(atan2(e_y, e_x) - \theta\right) & \text{if } \left\|[e_x, e_y]^T\right\|_2 > d_{\min} \\ K_\omega \left(r_\theta - \theta\right) & \text{otherwise} \end{cases} \tag{43}$$

Note that angular velocity control differentiates between positions far to the target (i.e. $\left\|[e_x, e_y]^T\right\|_2 > d_{\min}$) and those close to the target. In the first case, the controller tries to align the robot's orientation with the orientation of the error vector $[e_x, e_y]^T$ for a smooth trajectory. In the latter case, when the robot is close enough to the target, the controller tries to match the target orientation r_θ.

6.1.2 High-level Control

The main task of the high-level controller is to map the primitive motions that are computed by the self-reconfiguration algorithm to target poses that are fed into the low-level pose controller. This section describes the toolchain required to relay these target poses to the robots and close the position feedback loop through the overhead tracking camera. As shown in Figure 39, the main mode of interaction with the robots in the Robotarium occurs through a Matlab API - an interface class that enables the message exchange between the coordinating server application and the user (a comprehensive list of API functions is provided in Appendix B). The self-reconfiguration algorithm computes target poses in the workspace of the robots and submits these target pose requests to the server application. All

communication between the user application and the robots is handled by the server, which relays these target pose updates wirelessly to the respective robots. The pose controller outlined in Section 6.1.1 is executed directly on the robots. The feedback loop that enables pose control, however, is closed through the server and the tracking system. More specifically, the vision-based tracker detects the current poses of all robots and sends them through the server to the robots at approximately 25 Hz. The main difference between simulated self-reconfiguration and reconfiguration on hardware is the fact that the simulation treats primitive motions of agents as instantaneous. The execution on hardware is obviously bound by minimum execution times of these motions. Therefore, the self-reconfiguration algorithm has to wait until the respective robot finishes the execution of a motion.

The communication channels required to send target and current poses to the robots relies on two messaging interfaces. Communication between Matlab, the coordinating server application, and the tracker is based on the **L**ightweight **C**ommunications and **M**arshalling (LCM) library[1], which enables the exchange of structured messages. This library was chosen for interprocess communication for two main reasons. On the one hand, it requires little overhead in terms of resource usage and integration into the various software components of the framework. On the other hand, it requires no central node as the coordinating master node. LCM is implemented as a fully distributed publisher-subscriber message-passing framework. Though implemented very efficiently, LCM still requires too much memory to run on embedded hardware such as the main microcontroller on the GRITSBot. Therefore, the communication between the server and the robots relies on a different communication channel. Specifically, it uses WiFi networking and standard UDP sockets for message exchange.

6.1.3 Experimental Results

This section describes the setup and experimental results of two-dimensional game-theoretic self-reconfiguration on a team of eight GRITSBots. While the low-level controllers are executed on the robots themselves, the high-level self-reconfiguration algorithm runs in

[1]LCM is available at http://lcm-proj.github.io/

Matlab on the server. The basic decentralized algorithm (Algorithm 5) in connection with the adaptive learning rate as outlined in Section 5.1 accomplishes high-level control and decision making.

A Matlab script also performs the initial setup phase, i.e. moving all robots to their respective positions in the randomly generated initial configuration. To minimize the number of potential collisions during this setup phase, the Hungarian assignment algorithm computes an optimal assignment between positions the robots occupy at boot-up and their respective positions in the initial configuration. The Hungarian algorithm minimizes the cumulative distance traveled of all agents but potentially creates intersecting paths through its assignment. Therefore, in addition to the Hungarian algorithm, robots are equipped with basic obstacle and collision avoidance. After the setup phase is completed and agents start self-reconfiguring, collisions will not occur anymore due to the nature of the primitive motions of the sliding cube model. In other words, neither a sliding nor a corner motion will result in a collision given that grid cells are spaced sufficiently far apart. For the experiment shown in this section, neighboring grid cells were spaced 10 centimeters apart. This grid spacing has been chosen as a tradeoff between the available testbed surface area (120×80 cm) and the minimum distance between agents required to avoid collisions. A tighter grid may be possible but would complicate agent motion due to an increased number of potential collisions.

The experiment in this section shows a reconfiguration sequence from an initial random configuration to a rectangular configuration that is offset along the x-axis by one grid cells (an example is shown in the inset of Fig. 37). The initial and target configuration contained eight GRITSBot robots.[2] The self-reconfiguration sequence itself is equivalent to a simulation trial, the difference now being that target position updates are sent to the robots as well. The Matlab API briefly discussed in Section 6.1.2 bridges the gap between the self-reconfiguration simulator and the robotic hardware. Unlike in simulation, where a motion is assumed to be instantaneous, the robots' velocities are limited, which introduces a minimum execution time per motion primitive. Therefore, the high-level self-reconfiguration algorithm has to

[2]Eight robots are the currently maximum number of available robots.

Table 8: Numeric results for a self-reconfiguration sequence of eight robots.

Total number of time steps	[N]	371
Total number of motions	[N]	134
Total number of sliding motions	[N]	81
Total number of corner motions	[N]	53
Total distance traveled	[m]	15.59
Total reconfiguration time	[min]	15.46
Average time per move	[sec]	6.9

wait for the low-level controller to finish executing a motion before commanding another robot to start executing another action.

Simulation results suggest convergence times on the order of 250 to 350 time steps (see Fig. 37, shown is the average of 10 trial runs for each learning rate). The actual robotic experiment shown below required a total of 371 time steps to converge. Note however, that not all of these time steps represent agent actions. The majority of these time steps actually represent agents remaining at their current position without moving. According to the image sequence in Fig. 38 and the numeric results summarized in Table 8, out of 371 total time steps, only 134 represent agent motions for this specific hardware trial. Given that on average a motion takes 6.9 seconds to complete, even for such small configurations, the reconfiguration sequence lasted approximately 15.5 minutes. However, seven out of eight agents occupied target positions after only 6 minutes, while the remaining 9.5 minutes were required for the last agent to converge to its target position. In that sense, the hardware experiment confirmed the key characteristic of the game-theoretic self-reconfiguration approach: the initial fast ramp-up in global potential and the slower convergence to the global potential maximizer. As the number of modules in a system increases, this initial steep increase in potential will become more important compared to the eventual convergence to the full assembly of the target configuration. That is because even though not every module is at a target position, the configuration will closely resemble the target configuration in shape and function after the initial ramp-up already.

The experimental results presented in this section relied on a hardware setup that has been custom-designed and built for swarm-robotic experiments. As mentioned before, eight

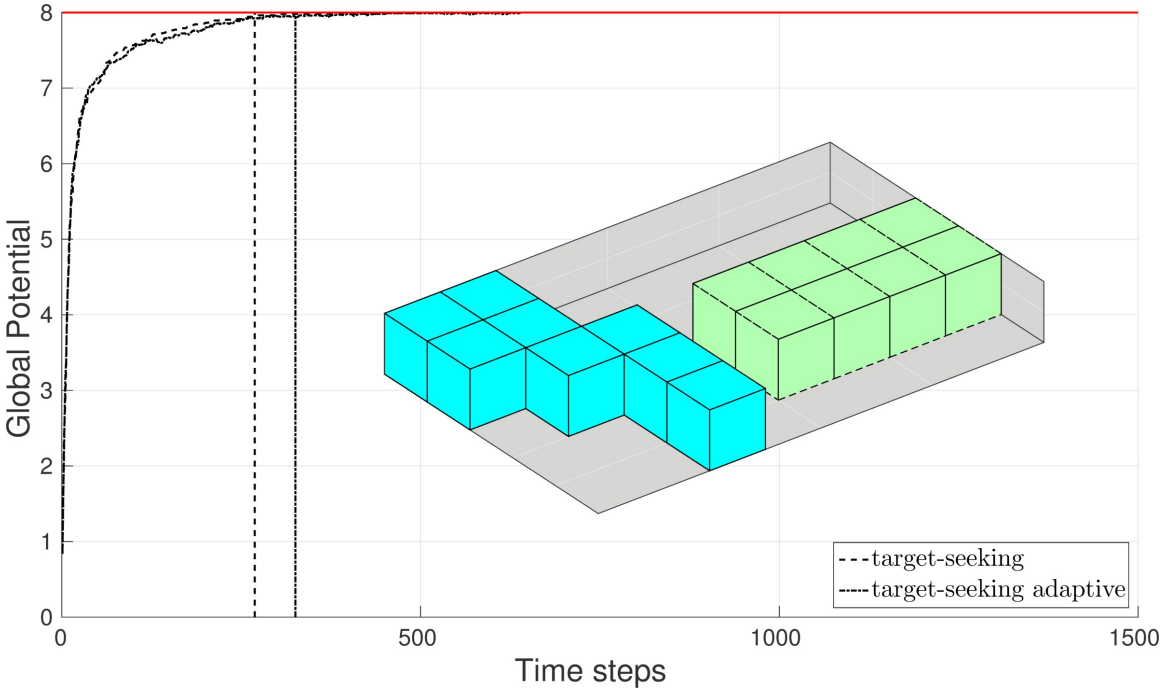


Figure 37: Average convergence time for a two-dimensional configuration of eight agents based on 10 trials.

GRITSBot robots have been used. These robots are embedded in a novel open-access multi-agent testbed called the Robotarium. The following sections will detail the concept of the Robotarium, the design requirements imposed on such a shared research instrument, and the design evolution of both the Robotarium testbed and the GRITSBots it contains.

6.2 The Robotarium Concept

Multi-robot research has seen a considerable growth during the last decade, with a number of coordinated control algorithms being developed for tasks ranging from environmental monitoring (e.g. [48, 49, 201]) to collective material handling (e.g. [143]). This growth has been driven by a combination of algorithmic advances, increased hardware miniaturization, and cost reduction. However, despite the reduction in cost, it is still a prohibitively costly proposition to move from theory and simulation, via a few robots, all the way to the deployment of a truly large-scale robot system. State-of-the-art experimental setups can cost tens of thousands of dollars in hardware alone, while the cost of maintaining and operating such testbeds can exceed the initial price tag. That is why currently there are only a

Figure 38: Image sequence of a self-reconfiguration trial on a team of eight GRITSBots.

handful of laboratories around the world that can field massive numbers of robots in the air, underwater, or on the ground (e.g.[85, 125, 50, 157, 158]). To advance multi-robot research further, actual deployment is crucial since it is increasingly difficult to faithfully simulate all the issues associated with making multiple robots perform coordinated tasks due to the increased task complexity. At its core, the Robotarium is therefore a shared multi-robot testbed that aims at remedying these issues by enabling researchers to remotely access a state-of-the-art multi-robot test facility. A number of elegant remote-access robot systems have been developed in the past ([85, 50, 151, 37]). What makes the Robotarium different, however, is its explicit focus on supporting multi-robot research, as opposed to, for example, educational applications or single-robot systems.

As part of the Robotarium's mission of providing access to multi-robot testbeds, the underlying robotic architecture needs to fit as many applications as possible. This requirement of generality is why the testbed is based on wheeled miniature ground robots. Specifically we use the GRITSBot, a low-cost differential-drive miniature robot designed with similar capabilities as most-commonly used platforms in academia (see [144] and Fig. 35). This robot architecture was chosen because a number of tasks can be accomplished with generic wheeled ground robots, such as vehicle routing [6], coverage control [7], or collective exploration [142, 185]. Therefore, the GRITSBot allows for a straightforward transition from current experimental setups to a GRITSBot-based system and enables researchers to set up a multi-agent testbed that resembles current state-of-the-art platforms at a fraction of the cost. In this sense, the Robotarium aims at providing not just an affordable, accessible, and user-friendly but also a flexible testbed that can be easily replicated. The general structure of the Robotarium infrastructure is shown in Fig. 39. Note that two approaches to remote access are currently implemented. Track 1 relies on the direct interaction of a user with the Robotarium through a Matlab or Python API. This allows both a centralized or decentralized execution by sending velocity or target position commands to the robots through a server application. Alternatively, users can submit code to the Robotarium server that will be verified, compiled, and uploaded to the robots. While code upload allows to close the feedback loop locally and has the potential to improve the performance of delay-sensitive

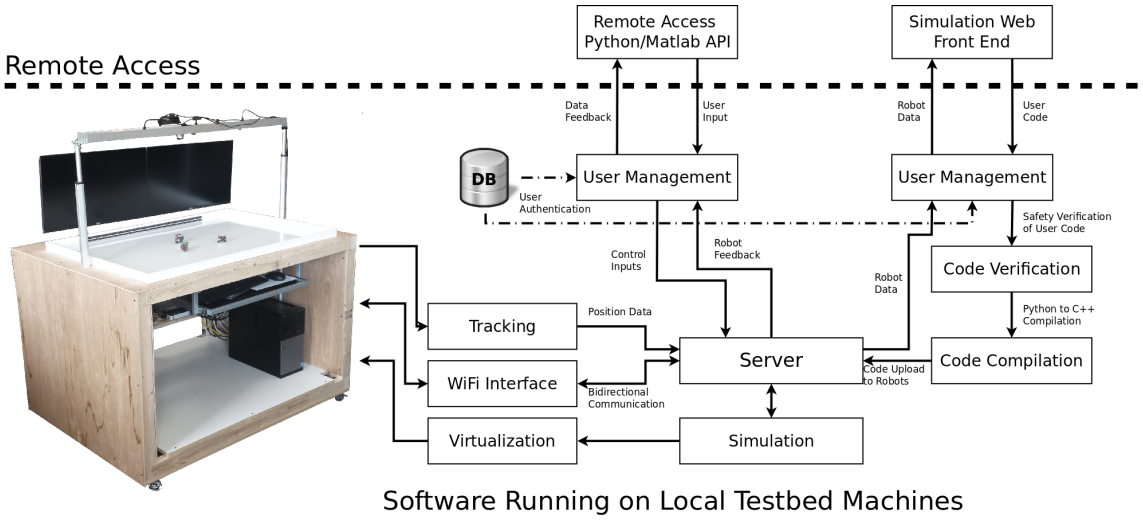


Figure 39: System architecture overview. The current prototype includes components that are executed locally on Robotarium infrastructure as well as user-facing components that run on remote user machines (APIs or simulation front end). Three components interact directly with the robot hardware - tracking, wireless communication, and virtualization. The remaining components handle user management, code verification and upload, as well as coordination of user data and testbed-generated data.

applications, this second track is currently still a manual process. The current web front end for code upload is still in its infancy and not fully integrated into the Robotarium. Note however, that the shown prototype of the Robotarium is fully functional and capable of executing multi-robot experiments. In fact, algorithms as varied as consensus/rendezvous, formation control, leader-follower networks (with and without connectivity maintenance), coverage control, circular path following, and self-reconfiguration have been successfully implemented on the Robotarium hardware. The next section discusses the design requirements that enable remote interaction and addresses usability, safety, and security challenges.

6.3 Design Requirements

As a shared, remotely accessible, multi-robot facility, the Robotarium's main purpose is to lower the barrier to entrance into multi-agent robotics. Similar to open-source software that provides access to high quality software, the Robotarium aims at providing universal access to state-of-the-art robotic infrastructure and enabling fast prototyping of multi-agent and swarm robotic algorithms. The Robotarium was designed primarily as a research and

educational tool that will function as a gateway to higher STEM education on the one hand and as an accessible and capable research instrument on the other hand. As such it is conceivable that a pervasive robotic testbed such as the Robotarium will have to exhibit a subset or all of the following high-level characteristics to fulfill its intended use effectively.

- Simple and inexpensive replicability of the system - both the testbed itself and the contained robots.
- Intuitive interaction with and data collection from the testbed.
- Tight and seamless integration of the simulation workflow for algorithm prototyping and code execution on the robots.
- Minimization of both cost and maintenance effort while keeping the robots and testbed extensible.
- Built-in safety and security measures to protect the system from damage and misuse.

These desired high-level characteristics can be mapped onto more specific constraints that inform the hardware design as well as the software architecture - both low-level controls on the robots as well as the coordinating server applications. An overview of our current instantiation of such a remote-access multi-robot testbed is shown in Fig. 39. Whereas this implementation already serves as a fully functional small-scale prototype, a full-fledged Robotarium implementation should include the following features.

- Large numbers of low-cost robots (on the order of hundreds, see Section 6.3.1)
- Convenience features to simplify maintenance of large collectives of robots (see Section 6.5)
- Immersive user-experience through a fully remotely accessible testbed with live video and data streaming (see Section 6.3.2)
- Public interface to allow users to schedule time on the testbed and get assigned a number of robots for use (see Section 6.3.3)

These design requirements can be categorized along three dimensions: robot design, user-experience, and network design.

6.3.1 Robots

Several hardware implementations have been proposed to serve as inexpensive robots for multi-agent experiments and experimental verification of algorithms for collective tasks. For example, self-reconfiguration has been executed on the M-blocks system [154], 2D self-assembly and collective transport have been implemented on the Kilobots [157][158], and collective construction has been verified on the TERMES system [143]. Most of these hardware platforms, however, have been tailored for use in a specific setting. A number of collective tasks, however, can be implemented using wheeled ground robots, for example, vehicle routing [6], coverage control [7], or collective exploration [142]. Even biologically inspired research such as the development of tracking algorithms for animals (for example ants) could be studied on the Robotarium [92]. Therefore, as varied as these research domains and results are, the systems used for implementation and verification of theoretic results are similar in most research labs - wheeled differential drive ground robots and optionally a motion capture system to track the position and orientation of robots. A number of wheeled ground robots are commercially available and used for research purposes - some of the most prevalent include the Khepera[3], the e-Puck (see [16]), or the r-one robots (see [121]).

The Robotarium, however, is meant to provide a well integrated, immersive user experience with the smallest possible footprint, and features that allow a large swarm to be maintained effortlessly. Such tight integration is only possible with custom hardware. Therefore, our custom-designed robots, the GRITSBots (see [144]), form the core of the Robotarium. These miniature robots ensure that the user community is not limited to simulating robots locally, but is also able to deploy its own low-cost, high-performance robots in conjunction with the Robotarium's robots - robots that already integrate the remote-access aspect as a key characteristic of their design. With ease of deployment in mind, we have designed the

[3]http://www.k-team.com/mobile-robotics-products/khepera-iii

GRITSBots to be user-friendly, simple to maintain, and to tie in seamlessly with the Robotarium. Overall, the goals for the GRITSBot are twofold: to remove the barrier to entry by significantly lowering the price tag of a multi-robot testbed and to make swarm robotics accessible and user-friendly for a wider research community by enabling a straightforward transition from current experimental setups to the GRITSBot.

Generally, the required capabilities and specifications of a testbed are determined by the algorithms that its robots are tasked to execute. The decentralized nature of a variety of multi-agent algorithms (such as rendezvous, formation control, vehicle routing), at a minimum, require local sensing and accurate locomotion. Furthermore, to broaden the scope of such a robotic testbed, it should offer the capability of remotely operating robots or closing the feedback loop through a remote machine. To enable automated maintenance tasks, some form of global communication and positioning system is also required. In summary, a robot used for inexpensive multi-robotic testbeds and swarm experiments should have at least the following capabilities.

- High resolution and accuracy locomotion
- Range and bearing measurements
- Global positioning system
- Wireless communication with a global host
- Small footprint

Furthermore, as robots are scaled down in size and scaled up in numbers additional maintenance and usability features become indispensable. These features allow a single user to easily handle large numbers of robots without the need to individually operate, program, charge, or calibrate them. These convenience features significantly speed up the experimental process and simplify the maintenance of a large collection of robots.

- Automatic sensor calibration
- Autonomous battery charging

- Wireless programming
- Local communication between robots

While the above outlined specifications presented desired hardware features of the robot and the testbed, a number of high-level design principles guided the development of the GRITSBot.

Simplicity Commercial availability is an advantage for those labs not equipped to assemble miniature robots. However, it introduces a significant markup over the cost of parts alone. The GRITSBot was designed with ease of assembly in mind. Therefore, the total number of surface mount devices (SMD) components per board was kept to a minimum. The motor board contains 11 SMD parts, the main board 32, and the sensor board 45. Therefore, not counting the header pins that connect the individual boards, the total part count comes to just 88 components.[4] Outsourcing the assembly to dedicated circuit board manufacturers, a robot can be fully assembled within 15 to 30 minutes since only the connectors and motors need to be soldered on. A comparison of parts cost alone versus assembly cost of the boards through a dedicated board manufacturer is shown in Tables 10 and 11.

Modularity Multi-robot systems can be used in a variety of settings each with specific requirements regarding sensing, actuation, and processing. Adaptability to environmental constraints and functional requirements dictates a modular design. For example, certain experiments might not require sensing (see Section 6.1.3), in which case it should be simple to reduce the robots functionality. The layered design of the GRITSBot allows for fine-grained adaptability of its functionality by removing or adding layers. Removing the sensor board, for example, would allow to reduce the weight of the robot and increase its battery lifetime. An additional benefit of this modular design is that users are then enabled to build their own custom layers to extend the robots functionality.

[4]By comparison, the Kilobot uses 78 parts (based on the public bill of materials) but is arguably not as versatile and capable a platform as the GRITSBot.

Scalability In simulation, multi-robot experiments can contain thousands or even millions of robots. However, typical hardware implementations are limited to at most hundreds of robots [34][91] or in the case of the Kilobot to 1024 [157]. Depending on the required capabilities, certain limitations are imposed on the number of concurrently operating GRITSBots as well. On the one hand, the field of view of the overhead camera limits the total size of the environment (if absolute positioning is required). On the other hand, the bandwidth of the WiFi channel limits the total number of concurrently operating robots (if global communication is required). In typical experiments, data is exchanged with a single robot at a rate of approximately 1 - 5 KB/sec. Given the 54 MBit/s bandwidth offered by the GRITSBot's WiFi chip, a conservative estimate puts the total number of supported robots at 1350 robots per WiFi channel. Typical WiFi routers offer between 4 (non-overlapping) and 14 (overlapping) channels in the 2.4 GHz band that is used by the GRITSBot.[5] Given that the number of robots does not seem to be limited by the communication bandwidth for any practical purpose, an upper limit will be imposed by the available testbed area and the maximum supported density of robots. This maximum density will vary depending on the executed algorithm, but current experiments suggest that densities of 20 to 30 robots per square meter represent a soft upper limit for typical swarm algorithms. Self-reconfiguration, where robots can be more tightly packed, potentially supports upwards of 30 robots per square meter.

Low Cost A major barrier for the widespread adoption of multi-agent testbeds is their prohibitive cost and to a lesser extent their size (of individual robots and of the full testbed setup). Whereas commercially available, wheeled robots are being sold at prices as low as $99 (e.g. the 3pi robot by Pololu, see Table 1 in [121]), few of these low-cost platforms are viable for research (for a number of reasons including a lack of required sensors, a too large footprint, too low accuracy of motion, difficulty of integration of large numbers of robots). Robots such as the e-Puck and the Khepera III are fully capable and assembled research

[5] Channel specification according to the IEEE 802.11 standard available at http://standards.ieee.org/about/get/802/802.11.html

platforms, however their price limits their use to well funded labs (see Table 9). On the lower end of the price spectrum one finds the Jasmine robot (see [91]), the Alice robot (see [34], [34]), the R-One (see [122]), and the newest addition, the Kilobot ($14 to $50 in parts depending on order quantities or $115 fully assembled, see [157]). With the exception of the Kilobot, these robots are neither commercially available nor are their designs fully disclosed or available for replication anymore. The GRITSBot is fully open-source and available online and a single robot can be built for approximately $40 (or $70 pre-assembled, see Tables 10 and 11).

Small Form Factor Available space in terms of room size often restricts the total number of robots in multi-agent experiments to a few dozens of robots (for example [16], [121]). Recently a lot of work has been dedicated to miniaturizing robots to the extent where a testbed fits on a table (see [34] or [157]). This miniaturization enables a much larger audience to participate in multi-robot experiments at a fraction of the cost and with much lower space requirements compared to previous hardware implementations. The GRITSBot features a footprint of 31×31 millimeters, which is approximately the size of a Kilobot. This small size allows swarm experiments with dozens of robots on testbeds as small as a few square meters.

Usability Since ease of use was one of the main design requirements for the GRITSBot, tools for setting up and maintaining a collective of miniature robots were required. As indicated in [157], such convenience features include collective programming, powering and charging, as well as collective control. An additional tool we developed was automatic sensor calibration (see Section 6.5). All these tools aim at automating the menial tasks of maintaining a large collective of robots by minimizing physical interaction with them. For example, an EEPROM chip on the GRITSBot enables wireless programming of both the motor and the main board. In addition to individually reprogramming a robot based on its unique wireless IP, it is also possible to broadcast reprogram all available robots or groups of robots connected to the same subnetwork.

Table 9: An Overview of Multi-Robot Platforms

Robot	**Cost**	**Scalability**	**Odometry**	**Sensors**	**Locomotion**	**Size [cm]**	**Battery [h]**
GRITSBot	$\$\sim 50$[1]	charge, program calibrate	stepper motors	distance, bearing, 3D accel., 3D gyro	wheel, 25cm/s	3	1-5
Kilobot [157]	$\$50$[1,2,4]	charge, power program	other agents	distance, ambient light	vibration, 1cm/s	3.3	3-24
Jasmine [91, 88]	$\$130$[1]	charging	wheel encoders	distance, bearing, color	wheel, 50cm/s	3	1-2
Alice [34, 33, 35]	N/A	none	wheel encoders	distance, bearing, cliff	wheel, 2cm/s	2.1	1-10
r-one [122]	$\$220$[1]	none	wheel encoders	visible light, 3D accel., 2D gyro, bump, IR	wheel, 30cm/s	10	6
SwarmBot [121]	N/A	charge, program, power, calibrate	wheel encoders	range, bearing, camera, bump	wheel, 50cm/s	12.7	3
e-puck [16]	$\$979$	none	wheel encoders	range, bearing, 3D accel. microphones	wheel, 13cm/s	7.5	1-10
Khepera III[3]	$\$2750$	none	wheel encoders	distance, bearing, IR ground sensors	wheel, 50cm/s	13	1-8

[1] Cost of parts
[2] Note that this price refers to order quantities of 100 or fewer
[3] Available for purchase at http://www.k-team.com/mobile-robotics-products/khepera-iii
[4] Available for purchase at http://www.k-team.com/mobile-robotics-products/kilobot for $1150 for 10 robots

6.3.2 User Experience

Being an integrated research instrument, the user experience needs to be a vital part of the design of a testbed such as the Robotarium. On the instrumentation side, the Robotarium is equipped with cameras that provide a video stream of the experiments, tracking cameras for localization, and projectors for adding virtual robots to the Robotarium floor that behave as if they were actual physical robots. These virtual robots enable interaction with other virtual and physical robots alike, where such interactions include both collision and obstacles avoidance.

6.3.3 Network Design

The shared nature of the Robotarium requires precautions to be taken against unauthorized access or abusive use of the system. Access to such a testbed will therefore have to be managed through a user verification and authentication system (for example LDAP in combination with SSH). Users will only be able to access the robots they have been approved to use during their assigned time slot. These access control mechanisms ensure security for the Robotarium by managing outside threats. The current prototype guards against these threats by only allowing users to access the Robotarium through a local wireless network. Alternatively, remote code submission and upload to the robots is possible but currently still a manual process. In the future, code upload will rely on secure access mechanisms, user authentication, and user vetting.

Closely linked to security is the safety aspect of the system, i.e. ensuring that the system does not damage itself. Therefore a vital component of the Robotarium's software architecture will be formal code verification to guarantee the avoidance of damage to the hardware through faulty, corrupted, or malicious code. The current prototype does not use formal code verification and instead relies on simulation-based verification. Before user code is allowed to be executed on the robots, it has to pass a simulation that verifies obstacle and collision avoidance. In addition, the velocities of the robots are limited to values that do not cause damage to the robots even when head-on collisions occur.

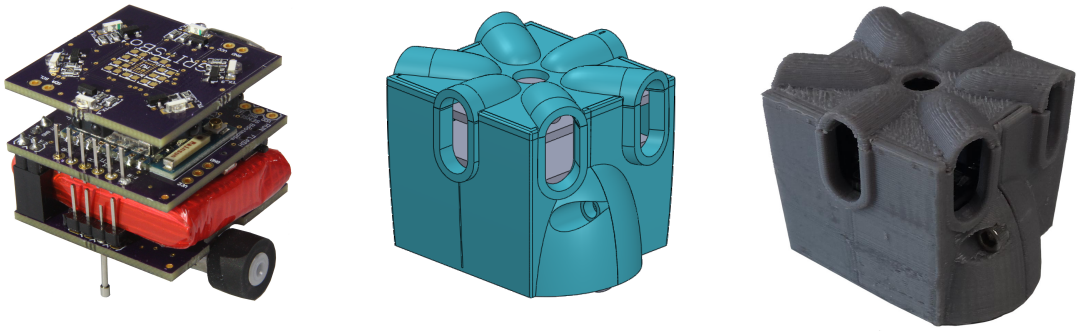
In addition to safety and security, network design has to take delay-tolerance into account. It is conceivable that a remotely accessible testbed has to accommodate user-testbed interaction on different timescales and with different delay tolerances. As shown in Fig. 39, two options for remote access are enabled by the Robotarium. Delay-insensitive applications and algorithms can remotely close the feedback loop through the provided APIs. This method will prove useful for quick prototyping and testing of algorithms that do not require high update rates, large amounts of data to be transfered, or a large number of robots to be involved. This use-case would apply to largely autonomous robots that require occasional user input to, for example, switch operating modes.

Delay-sensitive applications that require closing the feedback loop locally can make use of the second track of remote operation. User code is initially simulated on the user's local machine, and, after initial testing and verification, the code can be uploaded to the Robotarium. The code then undergoes simulation-based code verification and is executed locally on Robotarium hardware. Sample applications include large-scale robotic experiments or delay-critical applications that require robots to react quickly to sensor information.

6.4 The GRITSBot

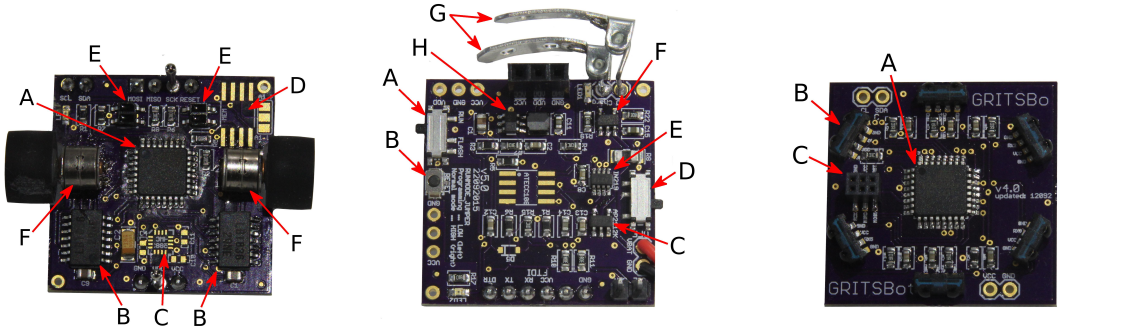
The GRITSBot is a novel differential drive miniature robot that features a layered design, where each layer fulfills a specific purpose and can be swapped in case of up-/downgrades or replacements (see Fig. 35 and Fig. 40a). This modular design was adopted for two main reasons: flexibility in adjusting the required capabilities of the robot to specific experiments and simplicity in design. This section describes each of the robot's functional blocks in detail and discusses how they are distributed across three circuit boards or layers. In addition, the design process is explained in detail and the design choices made throughout the development of the GRITSBot are justified.

The robot's main features include (i) high resolution and accuracy locomotion through miniature stepper motors, (ii) range and bearing measurements through infrared distance sensing, (iii) global positioning system and overhead camera system, and (iv) communication with a global host through a wireless transceiver. These features are discussed below and



(a) Electronics of the robot (b) 3D model (c) Prototype of robot shell

Figure 40: The current revision of the GRITSBot.



(a) The motor board layer (microcontroller (A), stepper motor driver (B), non-equipped magnetometer (C), non-equipped memory chip (D), line sensors (E), stepper motors (F))

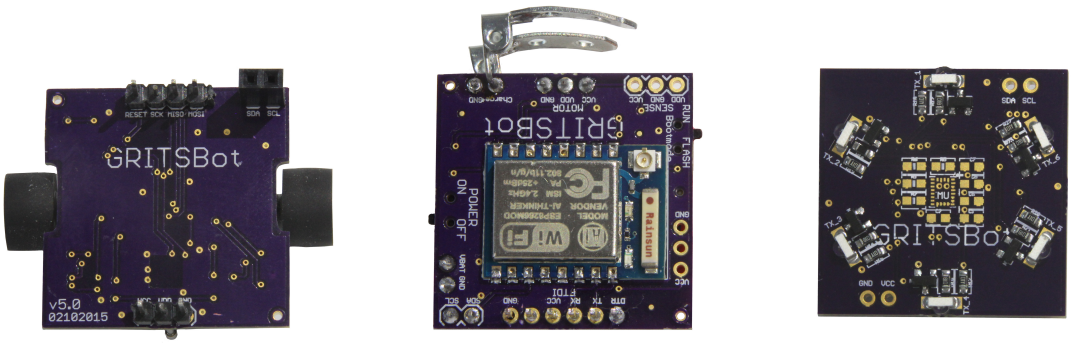
(b) The main board layer (power switch (A), reset button (B), voltage regulator (C), run mode switch (D), current sensor (E), battery charging chip (F), charging tabs (G), voltage step-up converter (H))

(c) The sensor board layer (microcontroller (A), infrared receiver (B), programming connector (C))

Figure 41: Bottom view of the three layers of the GRITSBot.

design changes compared to the first revisions of the robot in [144] are summarized.

Locomotion One of the novelties of the GRITSBot is its locomotion system. Unlike previous miniature robots, the GRITSBot does not use conventional DC motors and, therefore, does not require encoders to estimate their velocities. Instead, locomotion is based on miniature stepper motors. By their very nature, stepper motors completely obviate the need for velocity estimation since the target velocity of each motor can just be set through regulating the delay between individual steps. Odometry, therefore, is reduced to merely counting steps, which can be used to compute the velocities of the robot and estimate its



(a) The motor board layer (b) The main board layer (c) The sensor board layer

Figure 42: Top view of the three layers of the GRITSBot.

position.

Since encoders can introduce estimation inaccuracies, others have attempted to circumvent their use. A recent approach to encoder-free odometry has been proposed in [89]. In that implementation, however, complex signal processing is required to compute motor velocities. The Kilobot (see [157] and [158]) addresses odometry in a different way. Since its vibration motors do not allow for encoder-based odometry in the first place, the Kilobot estimates its position based on measured distances to stationary neighbors. A drawback of this approach is the dependence on other agents. In the design of the Kilobot, vibration-based actuation was chosen for cost reasons. However, the costs of the encoder-free stepper motor design of the GRITSBot are comparable[6], yet it offers high-accuracy locomotion at much higher top linear velocities of up to 25 cm/sec and rotational velocities of up to 820 degrees/sec.

Sensing A primary requirement of a miniature robot used in a multi-robot testbed is the measurement of distances and bearings to neighboring agents and obstacles. For reasons of both sensor size and cost, the GRITSBot, like most other miniature robots, relies on infrared-based (IR) distance sensing. Six IR transmitters and receivers are arranged in 60° increments around the circumference (see Fig. 41c(B)). These IR sensors are capable of

[6]The locomotion system of the GRITSBot costs $5.28 compared to $3.12 for the Kilobot (at quantities of 1000 robots). The cost estimates refer to quotes retrieved on February 1st, 2016 (see Appendix A).

measuring distances in the range of approximately 1 - 10 cm.[7]

In addition to IR sensing, the sensor board can be equipped with an accelerometer and gyroscope whose data can be fused into the velocity and position estimation to account, for example, for slip. Since few multi-agent experiments rely on acceleration and gyroscope data, this component is not equipped by default (according to Tables 10 and Table 11) but can be added to the robot for an additional $5.75. One more sensor that is mounted on the main board is a battery voltage and current sensor(see Fig. 41b(E)), whose data informs the control of the robot's autonomous charging behavior (see Fig. 6.5.4). The motor board can additionally be equipped with downward-facing infrared sensors for line-following applications (see Fig. 41a(E)) as well as a digital compass (see Fig. 41a(C)). The modular architecture of the robot allows to easily extend or change capabilities of the robot by replacing the sensor board with a custom board or simply stacking a second sensor board on top.

Processing The GRITSBot is equipped with two microcontrollers, an Atmega 168 chip on the motor board and an ESP8266 chip on the main board. Whereas the Atmega 168 chip (running at 8 MHz) is solely responsible for motor velocity control of the stepper motors, the ESP8266 chip (running at 80 or 160 MHz) is tasked with wireless communication, sensor data processing, low-level control of the robot (including the nonlinear velocity and position controller), and user-defined high-level tasks such as obstacle avoidance or other behaviors. Previous versions of the GRITSBot relied on the Atmega 328 chip as the main processing chip, but was upgraded to the more powerful ESP8266 chip that was also capable of built-in WiFi communication.

Communication The main ESP8266 microcontroller doubles as a WiFi transceiver supporting the IEEE 802.11 B/G/N standards. Unlike the wireless transceivers used on the GRITSBot in [144], WiFi offers much higher bandwidth but comes at the cost of higher

[7] For detailed specifications and parts lists see Appendix A.

power consumption (on average 150 mA). To offset the reduced battery life, we have increased battery capacity to 400 mAh compared to the 150 mAh used in [144]. The benefits of WiFi, however, far outweigh its increased power consumption. WiFi offers a reliable communication channel based on standard UDP sockets and a single WiFi access point is able to service hundreds of clients.

Previous versions of the GRITSBot were equipped with lower power RF transceivers. The first iteration of the robot was outfitted with HopeRF RFM69W transceivers[8] with a bandwidth of 300 Kbit/s and a power consumption in the range of 16 - 45 mA. These modules operated at 433 MHz or alternatively at 915 MHz. The relatively low bandwidth of these modules did not support high enough position or velocity update rates (in excess of 20 Hz) to large numbers of robots and was therefore abandoned for scalability reasons.

The second iteration of the GRITSBot employed Nordic Semiconductor nRF24L01+ transceivers[9] operating at 2.4 GHz. These low-power devices consumed below 15 mA even at their maximum transmission rate of 2 MBit/s. Initial experiments with small numbers of robots (on the order of two to three robots) showed that this bandwidth was sufficient for fast enough position or velocity updates to the robots. However, as more robots were added, either data packages were lost (in case no packet collision avoidance was used), or the transmission rate dropped significantly (when packet collision avoidance was used). For this reason, the robots were eventually equipped with far more capable WiFi modules, which both increased the available bandwidth and lowered the overall cost of the robot. The cost savings were achieved because the ESP8266 WiFi modules replaced both the nRF24L01+ transceiver and the Atmega328 microcontroller.

A desirable feature of a multi-robot testbed is certainly local communication. Whereas in principle the GRITSBot is capable of local IR-based communication, no such communication protocol is currently implemented. Note, however, that on the one hand, local communication is not required for most multi-robot experiments where distance and bearing measurements are available. On the other hand, local communication can be simulated

[8] RFM69W transceivers are available at http://www.hoperf.com/rf_transceiver/modules/RFM69W.html

[9] nRF24L01+ transceivers are available at https://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01P

through global communication to a host system should the need arise.

Power System The GRITSBot is powered by a 400 mAh single-cell lithium polymer (LiPo) battery that supplies a nominal voltage of 3.7V to the robot. The battery voltage is then regulated down to 3.3V - the system operating voltage - and stepped up to 5V - the voltage supplied to the motors. Both the power regulation and the battery charging circuitry are embedded into the main board, which supplies power to the motor and sensor board through header pins. The charging circuitry of the robot operates at 5V input voltage. When connected to a power supply, it charges the battery through a single-cell LiPo charging chip (see Fig. 41b(F)). The charging chip supplies up to 500 mA of current, which means the battery can be fully charged within approximately 45 minutes. Currently, depending on the activity level of the robot, it can operate between 40 to 60 minutes. Note that with the charging behavior in place the robot can recharge its battery autonomously, thus extending its battery life indefinitely. As shown in Section 6.5.4, the charging station is embedded into the testbed walls and, therefore, the robot can recharge without operator intervention.

Cost The costs per robot are based on order quantities of at least 50 robots, which results in parts cost of approximately 40$ per robot making the cost comparable to the Kilobot at low quantities. Table 10 summarizes the total parts cost by boards (assembly is not factored in).[10] The total cost of an assembled robot is shown in Table 11 and remains below 70$.[11] Note that assembly refers to populating the bare circuit boards with SMD components, which means that any through-hole components such as connectors and motors need to be soldered on by hand.

6.5 The Robotarium Testbed

The design of the GRITSBot and the Robotarium testbed allows a single user to easily operate and maintain a large collective of robots through built-in features such as (i) automatic sensor calibration, (ii) wireless (re)programming, (iii) automatic registration with

[10] Note that the design allows adding an IMU (gyroscope and accelerometer) for an additional 5.75$.

[11] The cost estimates refer to quotes retrieved on February 1st, 2016. The complete bill of materials including distributors is shown in Appendix A.

Table 10: Total parts cost per robot excluding assembly

Component	Cost	Function
Main board	16.44	Power management, WiFi, main processing, battery
Motor board	10.22	Actuation and motor control, motors, wheels
Sensor board	12.58	IR sensing, sensor data processing
Total	39.24	

Table 11: Total cost per robot including assembly

Component	Cost	Function
Main board	26.07	Power management, WiFi, main processing, battery
Motor board	18.24	Actuation and motor control, motors, wheels
Sensor board	24.48	IR sensing, accelerometer, gyro
Total	68.79	

the overhead tracking system, and (iv) autonomous battery charging. A possible extension that could significantly enhance multi-agent experiments is local communication, which the robot's sensor board supports with its dual-use infrared distance sensors. This section describes these convenience features in detail and provides insight into how they fit into the larger picture of remotely accessible testbeds.

6.5.1 Calibration

Since the robot measures voltages through its IR sensors and not metric distances per se, one has to establish the mapping between these voltages and the actual distance values in meters. The calibration station (see Fig. 43) provides such a mechanism and enables the automatic calibration of the robot's IR sensors. In case the calibration data is overwritten or corrupted, the robot can be recalibrated with minimal user intervention. The current model of the calibration station uses two stepper motors - one that rotates a platform holding the robot and one that linearly moves an obstacle. The rotating stepper motor ensures that only one of the robot's distance sensors is active and pointing directly at the obstacle. The second motor varies the distance of the obstacle to the robot in known increments which are then mapped to each of the corresponding sensor voltages. After this process is repeated for all six sensors, the configuration is written to the non-volatile EEPROM memory of the robot's main board. Therefore, the robot retains its calibration data and does not have to be recalibrated even after a power cycle. In the larger context of the Robotarium, automatic

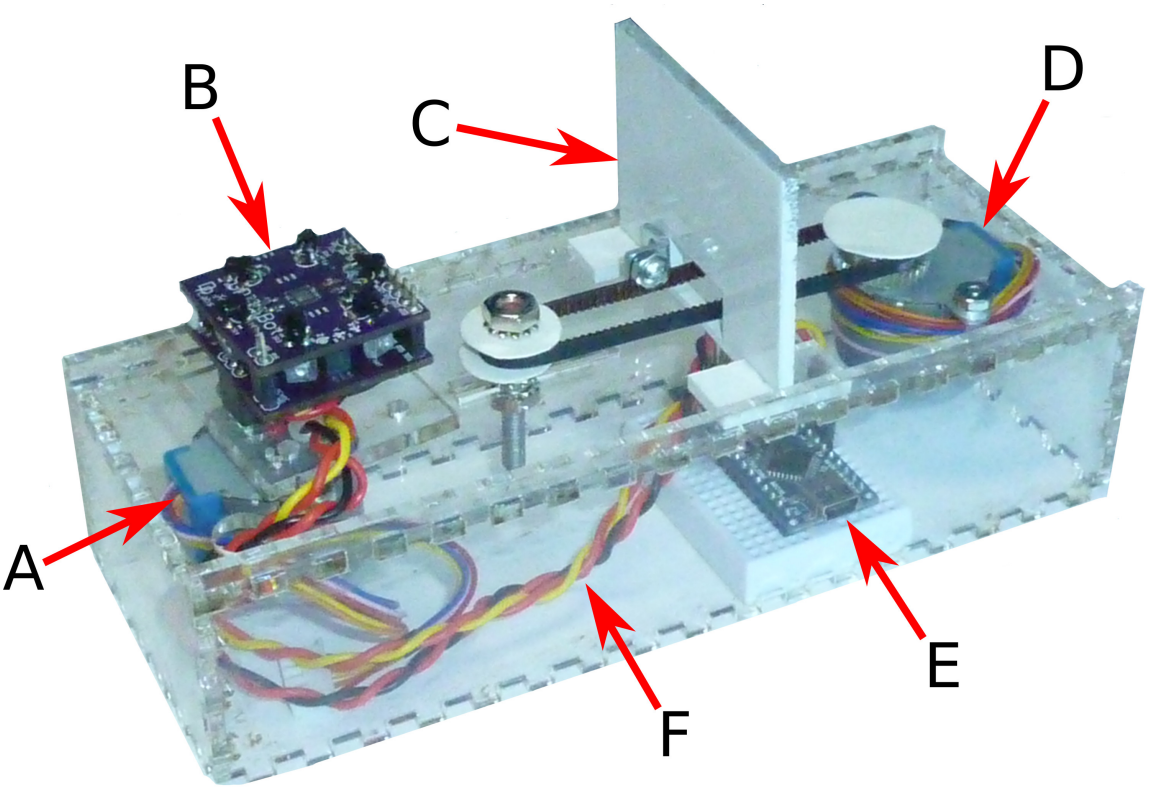


Figure 43: Automatic sensor calibration with the following components: (A) stepper motor rotating the robot platform, (B) a GRITSBot being calibrated, (C) controlled obstacle, (D) stepper motor controlling linear distance of obstacle to robot, (E) microcontroller, (F) communication and power supply between robot and calibration station.

sensor calibration ensures consistent sensor readings across multiple robots despite variations in the sensor hardware.

6.5.2 Wireless Programming

The main board's ESP8266 microcontroller supports over-the-air programming (OTA), which enables wireless reprogramming of individual robots, groups of robots, or even reprogramming a whole swarm in a broadcast fashion. It is even possible for one robot to reprogram another, which offers an array of research challenges in the domain of wireless security, as well as evolutionary and collaborative robotics. The Robotarium hardware enables OTA through the coordinating server application, which is capable of reprogramming robots over WiFi.

6.5.3 Global Positioning

The global position of all robots is retrieved through an overhead tracking system and is required to close the position control feedback loop, which the autonomous charging behavior of the robots depends on. Throughout the development of the Robotarium, the tracking system has undergone multiple revisions, which will be outlined below to highlight advantages and disadvantages of each approach as well as the reasons for switching to the current version using ArUco-based tracking (see [69]).[12] Note that for all tracking methods shown in this section that relied on tags, the tag size was limited to approximately 35 x 35 millimeters (the size of the robot footprint).

Most decentralized algorithms do not rely on global position updates but rather sensor data. Therefore, global positioning is not essential to their function. However, system maintenance such as recharging robots automatically or setting up an experiment (i.e. moving robots to user-specified positions) relies on globally accurate position data. As such some form of overhead tracking is key to the robust operation and maintenance of the Robotarium.

First Iteration - AprilTags-based Tracking This tracking method relied on QR-code-like identification tags called AprilTags (see [137] and Figure 44a).[13] Each robot was equipped with an identification tag and tracked using an overhead camera (more specifically, a Microsoft LifeCam Studio webcam). This computer vision-based overhead setup required image processing on a host machine and allowed the tracking of absolute coordinates as well as the orientation of the robots. The downside of having to equip every robot with an ID tag was outweighed by the accurate pose data that could be retrieved. However, the update rate proved to be too low to execute any fast robot motions accurately (linear robot velocities had to be limited to below 0.08 m/sec or approximately 30% of the maximum velocity). Even at relatively low resolutions of 800x600 using an Intel i7-4500U processor, the maximum achievable update rates averaged 10 Hz for up to five tracked robots and 5-7 Hz for up to 25 robots. Ultimately, the limited update rate was too restrictive to be useful.

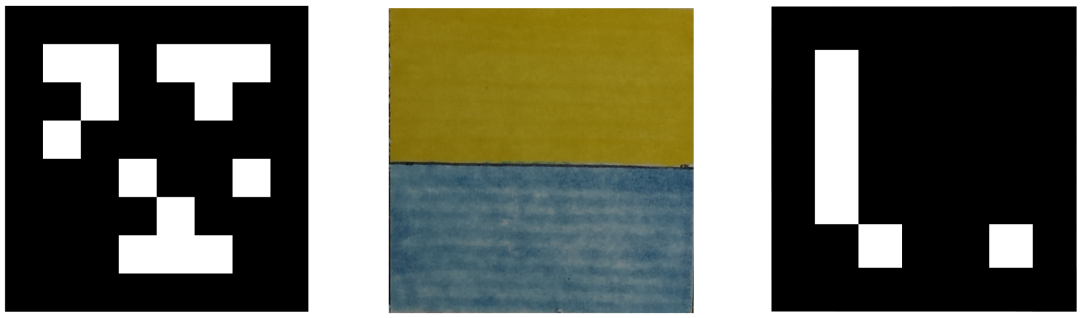
[12] ArUco is a minimal library for Augmented Reality applications based on OpenCV and can be found at http://www.uco.es/investiga/grupos/ava/node/26

[13] The AprilTags C++ library is available at http://people.csail.mit.edu/kaess/apriltags/

Second Iteration - Color Tag-based Tracking This tracking method relied on an integrated vision sensor called CMUcam5 Pixy [156], which was developed at Carnegie Mellon University and provided high-speed vision processing.[14] An update rate of 50 Hz was enabled by a dedicated hardware vision processing chip that was capable of tracking colored blobs as well as color tags (see Figure 44b). Using color tags, the system was able to quickly and accurately detect both the position as well as orientation of the robots. However, since the system relied on an accurate representation of color it was very susceptible to changes in lighting conditions. Whereas lighting conditions could be controlled, the main downside was the limited field of view of the camera. The small tag size of 35 x 35 mm required the camera to be located close to the robots it tracked (about 0.7 meters above the testbed floor). This small field of view would have required large numbers of cameras to cover the whole testbed area and was therefore unsuitable.

Third Iteration - Blob-based Tracking The third iteration of the overhead tracking system was based on blob tracking using a standard webcam (Microsoft LifeCam Studio HD camera). As such, no identification tags needed to be attached to the robots. The blob tracker was tuned to the color and size of the robots. While blob tracking allows the fast and efficient recovery of blob positions, the orientation of a blob cannot be recovered easily and requires additional post-processing using the motion model of the robots. The video stream was fed into an OpenCV-based blob tracking algorithm[15] which recovered time-stamped blob positions. These blob positions were associated with individual robots and their positions recorded over time. Position updates were then sent wirelessly to the robots and fed into a Kalman filter that computed orientation estimates in real-time. While the update rate of this approach was sufficiently high (in the range of 25 - 35 Hz depending on the number of blobs tracked and the background workload of the host machine), the convergence time of the Kalman filter limited the velocity of the robots and the aggressiveness of maneuvers that could be executed. Specifically, 180 degree-turns as well as turns on the spot were not

[14] The CMUcam5 Pixy is available at http://www.cmucam.org/projects/cmucam5.
[15] http://opencv.org/



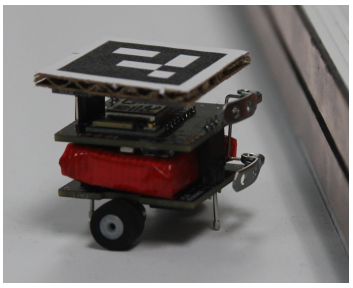
(a) Apriltags marker (b) Color tag marker (c) ArUco marker

Figure 44: Examples of markers used for position tracking.

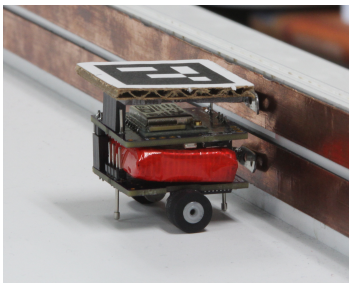
handled well and showed oscillations until the Kalman filter converged.

Fourth Iteration - ArUco-based Tracking The current iteration of the overhead tracking system relies on tools borrowed from the augmented reality community, specifically on a tag-based tracking method called ArUco (see [69]).[16] Automatically generated identification tags similar to AprilTags (see Figure 44a) have to be attached to each robot and can be tracked in real time using an efficient C++ implementation that relies on OpenCV - the de facto standard for computer vision applications.[17] The ArUco library provides well-documented code for the generation of tags and calibration boards, for camera calibration, and for the tracking of tags. In the conducted trials, the provided sample tracking code managed to maintain an update rate in excess of 25 Hz even for 200 tags present in the field of view of the camera. As is common for tag-based tracking method, ArUco retrieves both the position and the orientation of the tags it tracks - with millimeter accuracy after calibrating the camera. Besides accurate tracking, the main advantage is the large field of view that this approach supports despite the small 35 x 35 mm tags. The camera is located approximately 1.5 meters above the testbed floor, which allows the testbed to be covered by a single camera (see Figure 44c).

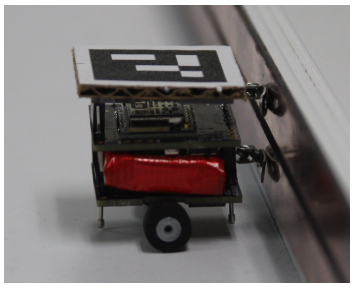
[16] ArUco is available at http://www.uco.es/investiga/grupos/ava/node/26.
[17] OpenCV is available at http://opencv.org/.



(a) GRITSBot approaching the charging station.



(b) GRITSBot connected to charging station.



(c) A side view of the GRITSBot connected to charging station.

Figure 45: The charging station for autonomous recharging of the GRITSBot's battery.

6.5.4 Charging

Arguably the most crucial component of a self-sustaining and maintenance-free testbed is an automatic recharging mechanism for the robots. The GRITSBot has been designed for autonomous recharging through two extending magnetic pins at the back of the robot that can connect to magnetic charging strips built into the testbed walls. One of the metal strips supplies a 5V input voltage while the other serves as ground. This setup together with global position control through the camera feedback loop allows the GRITSBot to autonomously recharge its battery (see Fig. 45).

In the larger context of remotely accessible testbeds, the charging behavior is the key aspect of the GRITSBot that will enable automated use of the robots and management of the Robotarium hardware without operator intervention. As part of the time scheduling mechanism, the Robotarium back end will not only assign a time slot to a user but also a number of available robots that are guaranteed to have been charged before the start of the experiment. After the conclusion of a user's experiment, the back end ensures that all used robots are returned to the charging station and prepared for the next time slot. Automating the recharging of robots is essential to making the continuous operation of the Robotarium economically feasible, especially as the number of robots is scaled up.

6.6 Conclusions

In this chapter, we have demonstrated how the self-reconfiguration algorithms introduced in Chapter 4 and 5 can be instantiated on robots. Specifically, we have shown how the discrete

motion primitives of the sliding cube model can be mapped to the continuous dynamics of the GRITSBots. Section 6.1 introduced velocity and position control laws that allow precise execution of these motion primitives in the continuous domain. A two-dimensional self-reconfiguration example established the feasibility of the algorithms presented in Chapter 4 and 5 through their execution on a novel testbed called the Robotarium.

The design and development of the Robotarium, a multi-robotic testbed geared towards remotely accessible and user-friendly multi-robot experiments, has been laid out in detail in this chapter. In particular, the Robotarium relies on the GRITSBot, a novel miniature robot designed to be the core component of the Robotarium. The GRITSBot is a highly capable wheeled research platform focused on low cost, simplicity in design, and ease of use. While being remotely accessible through the Robotarium, the GRITSBot can be easily replicated and employed in other research and educational institutions. A basic GRITSBot testbed can be set up with a handful of robots and a webcam used for tracking. The robot's design is fully open source and shown in detail in Appendix A, while the design process and iterations have been detailed in this chapter. As such, the GRITSBot's and the Robotarium's mission is to make multi-agent robotics available to a much larger target audience than it is today.

Chapter VII

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

The notion of self-reconfigurable systems was first introduced in the late 1980s by Fukuda et al. [68] who proposed applications in manipulation, mobility, and manufacturing. Since then self-reconfigurable systems have found their way into robotics, computer science, biology, chemistry, nanotechnology, and numerous other disciplines. Inspired by the assembly of biological cells in living organisms and the growth of organic and inorganic compounds such as crystals, the self-assembly of the future will rely on smaller and smaller modules. Regardless of the specific architecture of the basic building block of self-reconfigurable systems (whether organic or robotic), growing numbers of increasingly small modules in these systems underscore the importance of scalable approaches and a well-founded theoretic understanding of the workings of these systems. The research covered in this work aims at improving our understanding of self-reconfigurable systems, while moving the field of self-reconfigurable systems closer towards the deployment of truly large-scale systems.

After reviewing the current state-of-the-art of self-reconfigurable systems, this research began by developing centralized methods for self-reconfiguration planning. System models for both homogeneous and heterogeneous self-reconfigurable systems were defined, based on which provably complete reconfiguration planning algorithms were introduced. While the algorithmic results enabled a number of use-cases such as locomotion, self-(dis)assembly, and self-reconfiguration, the theoretic results served as the foundation for the decentralized methods in this work. Instead of computing motion plans on a central node, these decentralized methods enable modules to act as autonomous decision makers.

In particular, the main theoretic contribution of this research was then presented in the form of a game-theoretic formulation of the self-reconfiguration problem and a novel learning algorithm that provably solves it in a distributed fashion relying only on local information

and little communication between agents. Based on Metropolis-Hastings, this learning algorithm enables agents to weigh their probabilistic action selection in a way that provably guarantees the assembly of the target configuration. While capable of the same use-cases as the centralized methods, this agent-based formulation additionally enables scenarios such as splitting, merging, or avoiding obstacles in the environment. The key contribution of this approach is a theoretic framework that enables local, autonomous decision making with guaranteed global properties. The developed game-theoretic learning algorithm is suitable for a wide array of problems that rely on decentralized local control to guarantee globally emerging properties. While this decentralized approach already provides a number of attractive features for distributed execution, it is further extended to provide adaptability to the environment and improve convergence times. In particular, two approaches were presented that enable the runtime modification of the learning rate (used to adjust an agent's aggressiveness level towards exploration of the state space) and of the utility functions (used as incentivization for agents' action selection). Simulations show that for non-trivial scenarios, these methods have the potential to improve convergence rates compared to the basic non-adaptive method.

Finally, a complete multi-robotic testbed was developed including custom-designed miniature robots to validate these self-reconfiguration methods on actual hardware. An experiment executed by a team of eight robots showed the feasibility and utility of the game-theoretic self-reconfiguration approach. More importantly, this testbed - called the Robotarium - serves the much greater purpose of making multi-agent robotics accessible and affordable. Designed with a focus on remote-accessibility and user-friendliness, the Robotarium aspires to be to robotics what software as a service (SaaS) is to the software industry - a democratizing tool that makes state-of-the-art multi-agent robotics accessible to everybody.

In conclusion, this research investigated the self-reconfiguration problem from a modeling, algorithmic, theoretic, as well as hardware standpoint. By doing so, it not only provided deeper insights into the challenges and difficulties associated with each aspect of self-reconfiguration, but specifically addressed the issue of guaranteed convergence as well

as scalability through decentralized solution methods.

7.2 Future Work

The methods presented in this research provide a basis for the decentralized control of lattice-based self-reconfigurable systems. While theoretical as well as simulation and robotic results in this work confirm the feasibility and utility of a game-theoretic approach to self-reconfiguration, future work could investigate a number of open challenges related to game-theoretic self-assembly.

For example, one direction of future work could investigate the informational requirements of this approach. Currently, each agent is required to know the shape of the target configuration as well as its pose in the environment. Storing all positions of the target configuration in every agent's local memory may be feasible for systems on the order of thousands of robots. However, this strategy will have to be rethought as the number of modules in self-reconfigurable systems increases beyond millions of modules and/or the sizes of modules decreases. A possible solution might use approximation-based methods as shown in Section 2.1.2, for example, bounding box-based approaches.

Another future direction that could have major implications on the convergence rate is the notion of motion parallelism. Currently, agents take turns in executing actions. The resulting sequential nature of motions presents a limit to improving convergence rates. While the theory developed in Chapter 4 does not rule out parallel motions, the results will have to be adapted to guarantee convergence despite motion parallelism. However, the gains in convergence times could be significant.

Another possible way of improving convergence times addresses the asymptotic nature of the assembly of $\mathcal{C}_T$. In the current framework, using target-seeking utility functions does not inform agents which target positions are still unoccupied. Lacking this piece of information can cause agents to explore the vicinity of the target configuration for a large amount of time before eventually discovering the last remaining target positions. A possible future direction could explore methods of augmenting the data available to agents through communication with neighboring agents. For example, a gradient that guides agents towards unoccupied

target positions could be added to the game-theoretic framework for self-reconfiguration.

Lastly, the presented methods were developed for lattice-based systems, which characteristically rely on discrete and finite action sets. One major direction of future work is the extension of this approach to chain-type systems with their continuous and, therefore, infinite action sets. Can these continuous action sets be mapped to discrete action sets and handled by the presented algorithms without compromising the fluidity of motion of chain-type systems? Or alternatively, can the algorithms in this research be extended to be able to handle continuous action sets? While discretization might be able to represent infinite action sets in a feasible way, it is not immediately obvious how to incentivize the possible actions of chain-type systems. For example, a specific actuator joint angle might have to be awarded a much higher utility in one configuration versus another. The assumption of decoupled utility functions, which this research rests on, might have to be dropped to be able to model chain-type systems effectively.

When it comes to the Robotarium concept of remotely accessible multi-robot testbeds, the field is still largely unexplored. We are only at the beginning of understanding the needs, requirements, challenges, and opportunities associated with this new paradigm of making robots remotely accessible to the wider research community and eventually the general public. The primary research problems at this point concern infrastructure development, where the most challenging question is how to provide scalability and accommodate hundreds of mobile robots in real-time. The continuous and smooth operation of such a large-scale testbed will not only require reliable high-bandwidth wireless and wired connections but also a robust scheduling mechanism for assigning robots to users. As of right now, the most promising route to establish such scalability hints at web-based toolchains that support simulation, code verification, and eventually code upload to the physical assets of the Robotarium. Safety and security of the system will also pose major challenges. As an inherently shared and remotely accessed research instrument, the Robotarium is meant to be open-access. This open nature, however, will pose a significant challenge to address security needs in an effort to prevent misuse and malicious attacks on the one hand and safety, or the accidental damage to the system, on the other hand. Formal code verification or

simulation-based verification of algorithms could be used to diminish the risk of accidental damage, while proper user authorization and vetting will be potential methods to address the security needs.

Appendix A

ROBOT DESIGN

This appendix provides design files, bills of materials, as well as lists of manufacturers and distributors of parts. It provides specifics to facilitate sourcing of components and assembly of GRITSBot robots. In addition to the bill of material lists provided in Table 14, Table 15, and Table 16, Table 12 below summarizes details about all manufacturers and distributors used. Table 13 lists all miscellaneous components required for the assembly of the GRITSBot. Note that sourcing miniature stepper motors proved less reliable than sourcing all other electronic components. In general, 2-phase 4-wire stepper motors operating at 5V are required for the motor board. Miniature stepper motors with approximate dimensions of 4 - 6 mm in diameter and between 8 to 10 mm in length fit the design. One example of such a motor is listed in Table 13. This type of motor can also be found on Ebay as well as Amazon under the search term *miniature stepper motor*.

Table 12: List of distributors.

Distributor	**Website**	**Info**
Digi-Key Electronics	www.digikey.com	SMD electronics components
Mouser Electronics	www.mouser.com	SMD electronics components
Seeedstudio	www.seeedstudio.com	PCB manufacturing and assembly
OSH Park	www.oshpark.com	PCB manufacturing
Tower Hobbies	www.towerhobbies.com	LiPo batteries
Ebay	www.ebay.com	Stepper motors, ESP8266
Banggood	www.banggood.com	Connectors, ESP8266
Adafruit	www.adafruit.com	ESP8266 microcontroller
RobotShop	www.robotshop.com	Miniature wheels
Amazon	www.amazon.com	Charging pins

Table 13: Miscellaneous components used in the assembly of the GRITSBot.

Part Description	**Distributor**	**Part Number**	**Info**
Stepper motors	Banggood	981643	2-phase, 4-wire, 5V
Rubber wheels	RobotShop	RB-Sbo-09	10 mm outer diameter
LiPo battery	TowerHobbies	L5WLT902	3.7V, 400 mAh, 1S
Connectors	Banggood	1033759, 945516	Male/female header pins
Charging pins	Banggood	1011128	Brooch safety pins

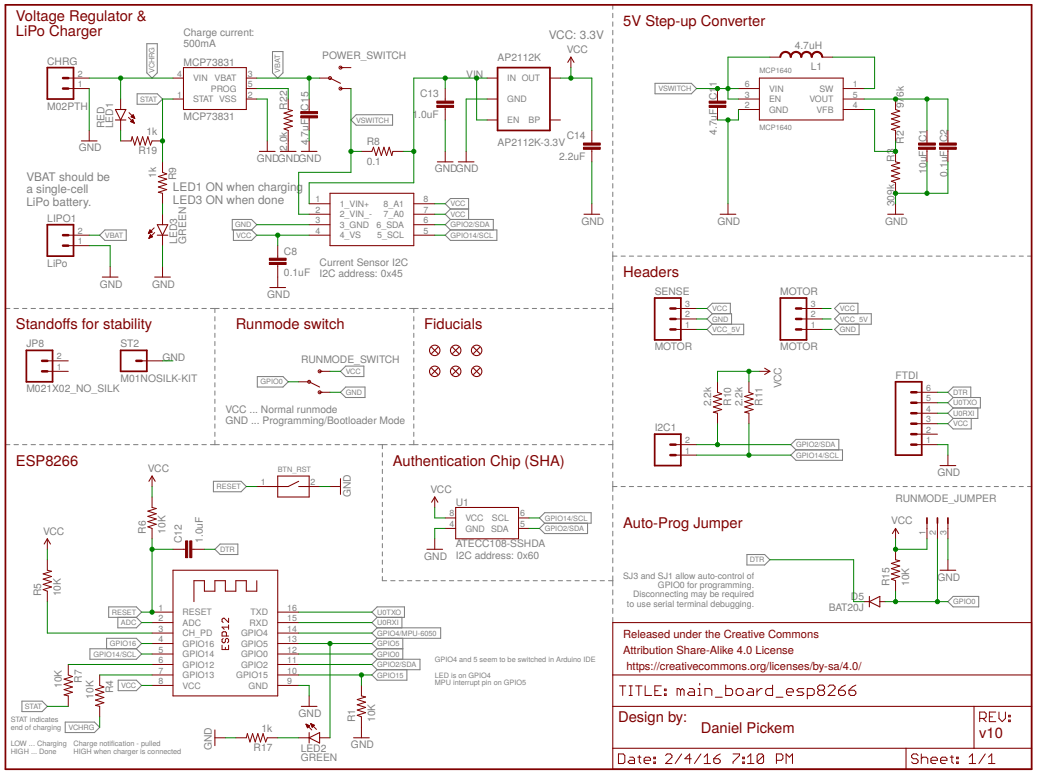


Figure 46: Schematic of the main board.

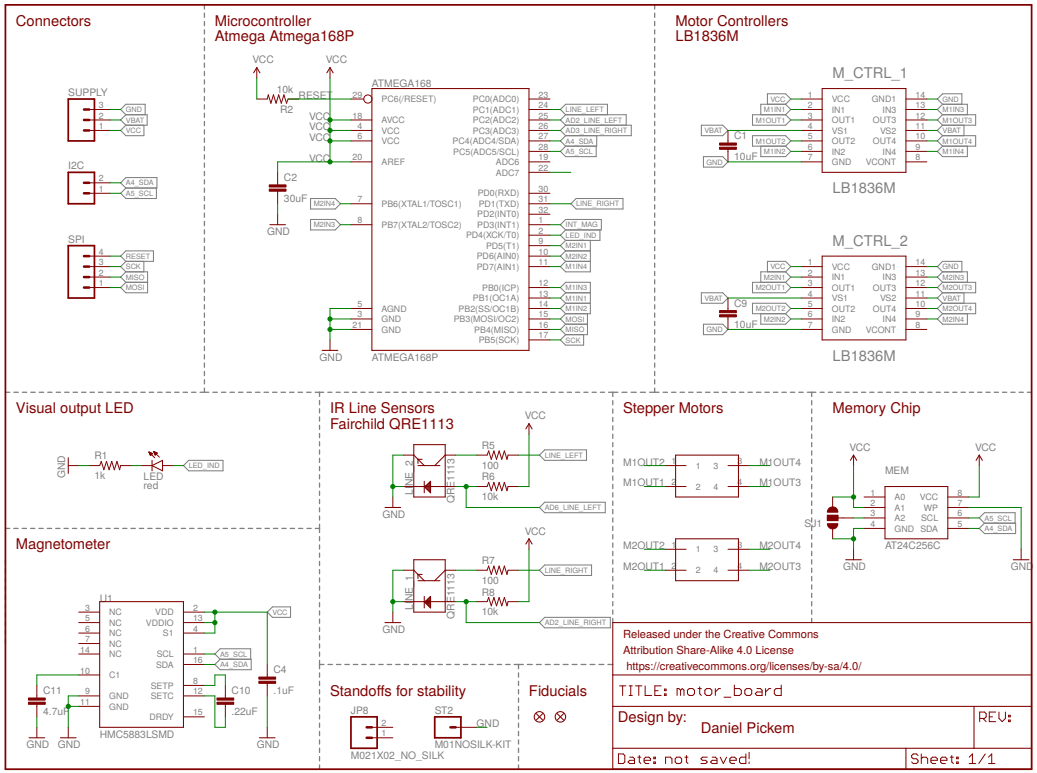


Figure 47: Schematic of the motor board.

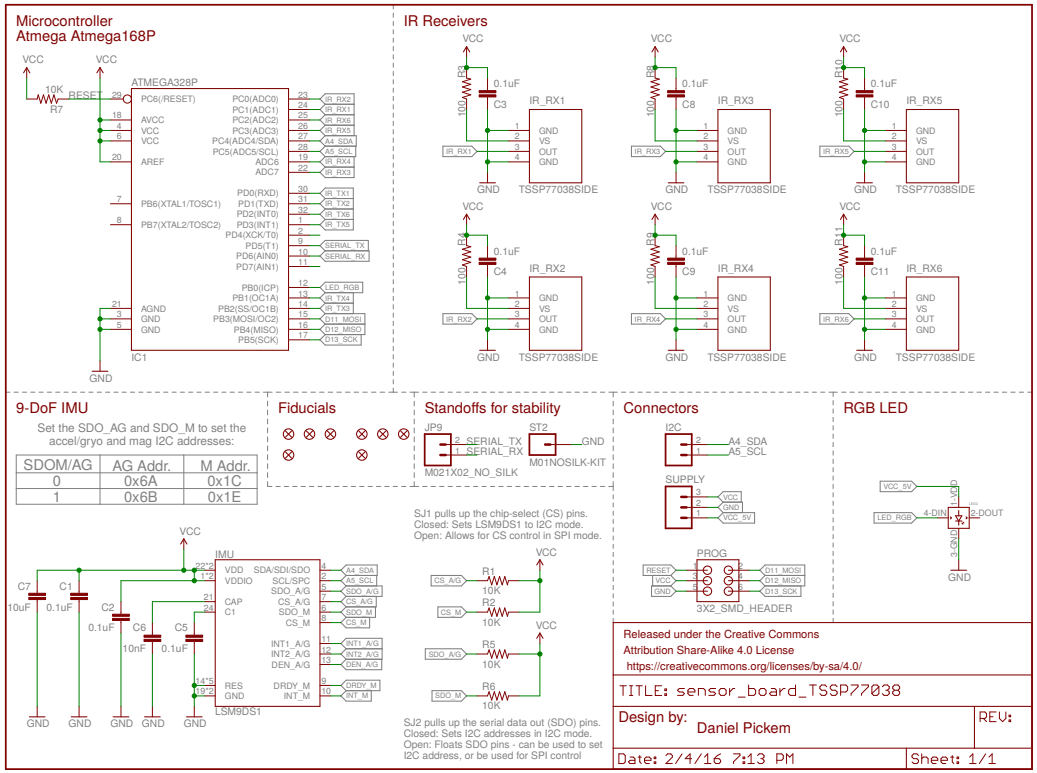


Figure 48: Schematic of the sensor board (part 1).

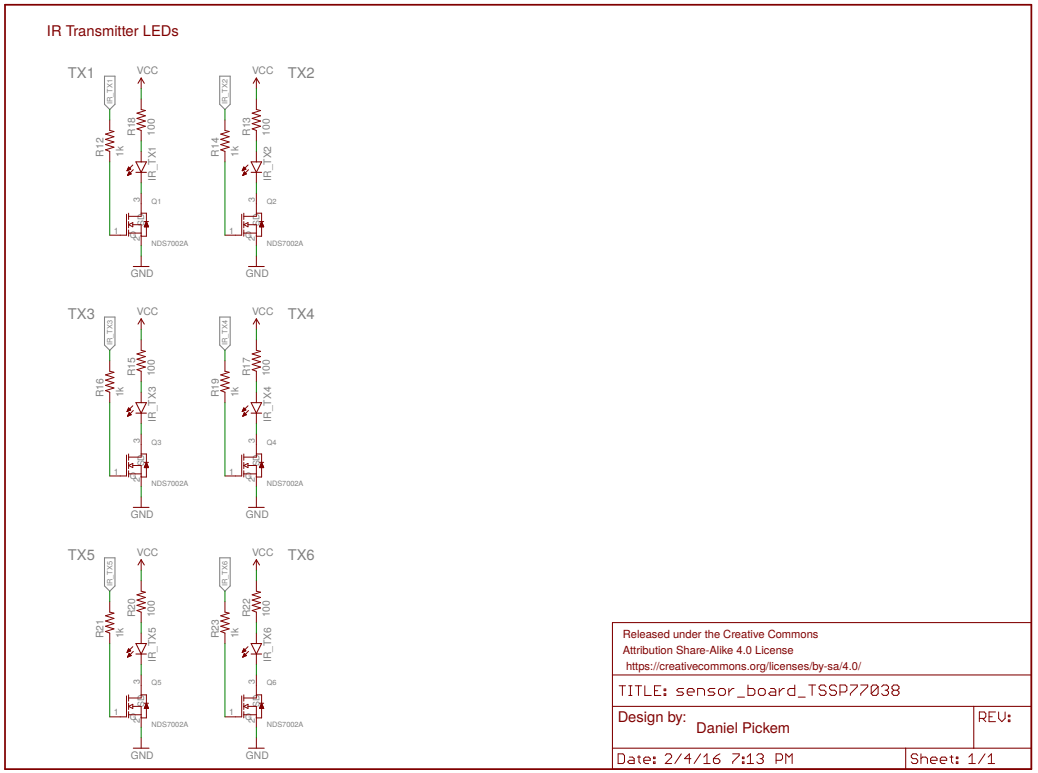


Figure 49: Schematic of the sensor board (part 2).

Table 14: Bill of materials of the main board (optional components highlighted in green).

Part Designator	Part Number	Quantity	Info	Distributor
AP2112K	AP2112K-3.3TRG1	1	Voltage regulator	Digi-Key
BTN_RST	PTS820 J15M SMTR LFS	1	Reset button	Digi-Key
C1	CC0805ZKY5V6BB106	1	10uF	Digi-Key
C2, C8	CC0603KRX7R7BB104	2	0.1uF	Digi-Key
C11, C15	CC0603KRX5R6BB475	2	4.7uF	Digi-Key
C12, C13	CC0603ZRY5V7BB105	2	1.0uF	Digi-Key
C14	CC0603KRX5R6BB225	1	2.2uF	Digi-Key
ESP8266	ESP8266 ESP-12E	1	Microcontroller + WiFi	Ebay
INA219	INA219AIDCNR	1	Current sensor	Digi-Key
L1	NR3015T4R7M	1	4.7uH	Digi-Key
LED1	5988020107F	1	red	Digi-Key
LED2, LED3	5988060107F	2	green	Digi-Key
MCP1640	MCP1640BT-I/CHY	1	Step-up converter	Digi-Key
MCP73831	MCP73831T-2ACI/OT	1	Lipo charging chip	Digi-Key
POWER, RUNMODE	PCM12SMTR	2	Switches	Digi-Key
R1, R4, R5, R6, R7, R15	RC0603FR-0710KL	5	10k	Digi-Key
R2	RC0603FR-07976KL	1	976k	Digi-Key
R3	RC0603FR-07309KL	1	309k	Digi-Key
R8	ERJ-8BWFR100V	1	0.1	Digi-Key
R10, R11, R22	RC0603FR-072KL	3	2.0k	Digi-Key
R9, R17, R19	RC0603FR-071KL	2	1.0k	Digi-Key
U1	ATECC108A-SSHCZ-B	1	Encryption chip	Digi-Key

Table 15: Bill of materials of the motor board (optional components highlighted in green).

Part Designator	**Part Number**	**Quantity**	**Info**	**Distributor**
ATMEGA168P	ATMEGA168PB-AU	1	Microcontroller	Digi-Key
C1, C9	CC0805ZKY5V6BB106	2	10uF	Digi-Key
C2	TAJA336K004RNJ	1	33uF	Digi-Key
C4	CC0603KRX7R7BB104	1	0.1uF	Digi-Key
C10	CC0603KRX7R8BB224	1	0.22uF	Digi-Key
C11	CC0603KRX5R6BB475	1	4.7uF	Digi-Key
LED	5988060107F	1	green	Digi-Key
LINE1, LINE2	QRE1113GR	2	IR line sensor	Digi-Key
M1, M2	Stepper Motors, LPD4	2	Stepper motors	Ebay
MEM	AT24CM01-SSHM-B	1	EEPROM memory	Digi-Key
M_CTRL_1, M_CTRL_2	LB1836M-TLM-E	2	Motor controller	Digi-Key
R1	RC0603FR-071KL	1	1k	Digi-Key
R2, R6, R8	RC0603FR-0710KL	3	10k	Digi-Key
R5, R7	RC0603FR-07100RL	2	100	Digi-Key
U1	HMC5883L-TR	1	Magnetometer	Digi-Key

Table 16: Bill of materials of the sensor board (optional components highlighted in green).

Part Designator	**Part Number**	**Quantity**	**Info**	**Distributor**
C1, C2, C3, C4, C5, C8, C9, C10, C11	CC0603KRX7R7BB104	9	0.1uF	Digi-Key
C6	CL10B103JB8NNNC	1	10nF	Digi-Key
C7	CC0805ZKY5V6BB106	1	10uF	Digi-Key
IC1	ATMEGA328P-AU	1	Microcontroller	Digi-Key
IMU	LSM9DS1TR	1	Gyro/Accelerometer	Digi-Key
IR_RX1 - IR_RX6	TSSP77038TT	6	IR receiver	Digi-Key
IR_TX1 - IR_TX6	VSMB10940	6	IR transmitter	Digi-Key
PROG	20021321-00006C4LF	1	Programming header	Digi-Key
Q1, Q2, Q3, Q4, Q5, Q6	NDS7002A	6	Transistor	Digi-Key
R1, R2, R5, R6, R7	RC0603FR-0710KL	5	10k	Digi-Key
R2, R3, R8, R9, R10, R11	RC0603FR-07100RL	6	100	Digi-Key
R12, R14, R16, R19, R21, R23	RC0603FR-071KL	6	1k	Digi-Key
R13, R15, R17, R18, R20, R22	RC0603FR-071RL	6	1	Digi-Key

Appendix B

ROBOTARIUM MATLAB API

This appendix provides a list of all functions the Matlab API provides for the control of robots in the Robotarium. Detailed inline documentation regarding the input and output parameters of each function is provided directly in the API implementation.

Table 17: List of functions the Matlab API provides (part 1).

Function Name
Constructor
function r = robotariumMatlabAPI(scenario, N)
function initializeScenario(r, scenario)
Communication primitives with back end
function sendMessage(r, IP, msgType, parameters)
function receiveMessage(r)
function s = requestData(r, IP, msgType, parameters)
GET functions
function p = getState(r, id)
function p = getTargetPose(r, id, fromRobot)
function v = getVelocity(r, id)
function [v, i] = getBatteryLevel(r, id)
function d = getIRDistance(r, IP, sensorID)
function d = getIRDistances(r, id)
function [N, X] = getNeighbors(r, id, maxDistance)
function [N, X] = getNeighborsFromTopology(r , id)
SET functions
function setVelocity(r, id, v)
function setVelocityMax(r, id, v)
function setTargetPose(r, id, pose)
function setSimulationMode(r, simMode)
GET GROUP functions
function N = getAvailableRobots(r)
function X = getStates(r)
function X = getPoses(r)
function V = getVelocities(r)
function P = getTargetPoses(r)
function [V, I] = getBatteryLevels(r)
SET GROUP functions
function setTargetPoses(r, poses)
function setVelocities(r, velocities)
function stopAllRobots(r)

Table 18: List of functions the Matlab API provides (part 2).

Function Name
Scenario-related GET functions
function F = getFormation(r)
function d = getDensity(r, x, y)
function d = getMaxDistance(r)
Scenario-related SET functions
function setMaxDistance(r, maxDistance)
function setTopology(r, adjMatrix)
function setDensity(r, n)
function setDensityFunction(r, densityFcnHandle)
function setDeltaDiskDistance(r, maxDist)
UPDATE functions
function update(r)
function updateDynamics(r)
function updateWaypoints(r)
DRAW / DISPLAY functions
function draw(r, axesHandle)
function updateEdges(r, Xe, Ye, axesHandle)
UTILITY functions
function n = N(r)
function [v, s] = saturateVelocities(r, vRaw)
function [T, B, L, R] = getDistanceToBoundaries(r , id)
function b = isWithinBoundaries(r, pose)
function b = isAtTarget(r, id, thresholds)
function b = allAtTarget(r, thresholds)

REFERENCES

[1] Ahmadzadeh, H. and Masehian, E., "Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization," *Artificial Intelligence*, vol. 223, pp. 27–64, 2015.

[2] Aicardi, M., Casalino, G., Bicchi, A., and Balestrino, A., "Closed loop steering of unicycle like vehicles via lyapunov techniques," *Robotics Automation Magazine, IEEE*, vol. 2, no. 1, pp. 27–35, 1995.

[3] Aloupis, G., Collette, S., Damian, M., Demaine, E. D., Flatland, R., Langerman, S., O'Rourke, J., Ramaswami, S., Sacristan, V., and Wuhrer, S., "Linear reconfiguration of cube-style modular robots," *Computational geometry*, vol. 42, no. 6, pp. 652–663, 2009.

[4] Aloupis, G., Collette, S., Damian, M., Demaine, E. D., Flatland, R., Langerman, S., O'Rourke, J., Ramaswami, S., Sacristán, V., and Wuhrer, S., "Linear reconfiguration of cube-style modular robots," in *Algorithms and Computation*, pp. 208–219, Springer, 2007.

[5] An, B. K., "Em-cube: cube-shaped, self-reconfigurable robots sliding on structure surfaces," in *Robotics and Automation (ICRA), 2008 IEEE International Conference on*, pp. 3149–3155, 2008.

[6] Arsie, A., Savla, K., and Frazzoli, E., "Efficient routing algorithms for multiple vehicles with no explicit communications," *Automatic Control, IEEE Transactions on*, vol. 54, no. 10, pp. 2302–2317, 2009.

[7] Arslan, G., Marden, J., and Shamma, J., "Autonomous vehicle-target assignment: A game-theoretical formulation," *Journal of Dynamic Systems, Measurement, and Control*, vol. 129, no. 5, pp. 584–596, 2007.

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

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

[10] Ashley-Rollman, M., Pillai, P., and Goodstein, M., "Simulating multi-million-robot ensembles," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 1006–1013, May 2011.

[11] Balch, T. and Parker, L. E., *Robot teams: from diversity to polymorphism*. AK Peters, Ltd., 2002.

[12] Bevan, M. A., Ford, D. M., Grover, M. A., Shapiro, B., Maroudas, D., Yang, Y., Thyagarajan, R., Tang, X., and Sehgal, R. M., "Controlling assembly of colloidal particles into structured objects: Basic strategy and a case study," *Journal of Process Control*, vol. 27, pp. 64 – 75, 2015.

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

[14] Bishop, J., Burden, S., Klavins, E., Kreisberg, R., Malone, W., Napp, N., and Nguyen, T., "Programmable parts: a demonstration of the grammatical approach to self-organization," in *Intelligent Robots and Systems (IROS), 2005 IEEE/RSJ International Conference on*, pp. 3684–3691, 2005.

[15] Blume, L. E., "The statistical mechanics of strategic interaction," *Games and economic behavior*, vol. 5, no. 3, pp. 387–424, 1993.

[16] Bonani, M., Raemy, X., Pugh, J., Mondana, F., Cianci, C., Klaptocz, A., Magnenat, S., Zufferey, J. C., Floreano, D., and Martinoli, A., "The e-puck, a robot designed for education in engineering," *Proceedings of the 9th Conference on Autnomous Robot Systems and Competitions*, vol. 1, pp. 59–65, 2009.

[17] Bonardi, S., Blatter, J., Fink, J., Moeckel, R., Jermann, P., Dillenbourg, P., and Ijspeert, A., "Design and evaluation of a graphical ipad application for arranging adaptive furniture," in *Robot and Human Interactive Communication (RO-MAN), 2012 IEEE International Symposium on*, pp. 290–297, 2012.

[18] Bonardi, S., Vespignani, M., Moeckel, R., and Ijspeert, A., "Collaborative manipulation and transport of passive pieces using the self-reconfigurable modular robots roombots," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pp. 2406–2412, 2013.

[19] Bonardi, S., Möckel, R., Spröwitz, A., Vespignani, M., and Ijspeert, A., "Locomotion through reconfiguration based on motor primitives for roombots self-reconfigurable modular robots," in *Robotik, 2012 7th German Conference on Robotics*, pp. 1–6, 2012.

[20] Boncheva, M., Bruzewicz, D. A., and Whitesides, G. M., "Millimeter-scale self-assembly and its applications," *Pure Applied Chemistry*, vol. 75, no. 5, pp. 621–630, 2003.

[21] Bourgeois, J. and Goldstein, S., "Distributed intelligent mems: Progresses and perspectives," *Systems Journal, IEEE*, vol. 9, no. 3, pp. 1057–1068, 2015.

[22] Boyd, S., Ghosh, A., Prabhakar, B., and Shah, D., "Randomized gossip algorithms," *Information Theory, IEEE Transactions on*, vol. 52, no. 6, pp. 2508–2530, 2006.

[23] Brandt, D. and Christensen, D., "A new meta-module for controlling large sheets of atron modules," in *Intelligent Robots and Systems (IROS), 2007 IEEE/RSJ International Conference on*, pp. 2375–2380, 2007.

[24] Brandt, D., Christensen, D., and Lund, H., "Atron robots: Versatility from self-reconfigurable modules," in *Mechatronics and Automation (ICMA), 2007 International Conference on*, pp. 26–32, 2007.

[25] Brandt, D., "Comparison of a and rrt-connect motion planning techniques for self-reconfiguration planning," in *Intelligent Robots and Systems (IROS), 2006 IEEE/RSJ International Conference on*, pp. 892–897, 2006.

[26] Brandt, D. and Østergaard, E. H., "Behaviour subdivision and generalization of rules in rule based control of the atron self-reconfigurable robot," in *Robotics and Automation (ISRA), 2004 International Symposium on*, pp. 67–74, 2004.

[27] Brookshear, J. G., *Theory of computation: formal languages, automata, and complexity*. Benjamin-Cummings Publishing Co., Inc., 1989.

[28] Butler, Z., Kotay, K., Rus, D., and Tomita, K., "Generic decentralized control for a class of self-reconfigurable robots," in *Robotics and Automation (ICRA), 2002 IEEE International Conference on*, vol. 1, pp. 809–816 vol.1, 2002.

[29] Butler, Z., Kotay, K., Rus, D., and Tomita, K., "Cellular automata for decentralized control of self-reconfigurable robots," in *Workshop on Modular Robots at Robotics and Automation (ICRA), 2001 IEEE International Conference on*, pp. 21–26, 2001.

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

[31] Butler, Z., Murata, S., and Rus, D., "Distributed replication algorithms for self-reconfiguring modular robots," in *Distributed Autonomous Robotic Systems (DARS), 2002 International Symposium on*, pp. 37–48, 2002.

[32] Campbell, J. D. and Pillai, P., "Collective actuation," *International Journal of Robotics Research*, vol. 27, no. 3-4, pp. 299–314, 2008.

[33] Caprari, G., Arras, K., and Siegwart, R., "The autonomous miniature robot alice: from prototypes to applications," in *Intelligent Robots and Systems (IROS), 2000 IEEE/RSJ International Conference on*, pp. 793–798, 2000.

[34] Caprari, G., Balmer, P., Piguet, R., and Siegwart, R., "The autonomous micro robot alice: a platform for scientific and commercial applications," in *Micromechatronics and Human Science (MHS), 1998 IEEE International Symposium on*, pp. 231–235, 1998.

[35] Caprari, G. and Siegwart, R., "Mobile micro-robots ready to use: Alice," in *Intelligent Robots and Systems (IROS), 2005 IEEE/RSJ International Conference on*, pp. 3295–3300, 2005.

[36] Casal, A. and Yim, M. H., "Self-reconfiguration planning for a class of modular robots," in *Intelligent Systems and Advanced Manufacturing, 1999 SPIE International Symposium on*, pp. 246–257, 1999.

[37] Casan, G., Cervera, E., Moughlbay, A., Alemany, J., and Martinet, P., "Ros-based online robot programming for remote education and training," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 6101–6106, 2015.

[38] Castano, A. and Will, P., "Representing and discovering the configuration of conro robots," in *Robotics and Automation (ICRA), 2001 IEEE International Conference on*, pp. 3503–3509, 2001.

[39] Castano, A. and Will, P., "Mechanical design of a module for reconfigurable robots," in *Intelligent Robots and Systems (IROS), 2000 IEEE/RSJ International Conference on*, pp. 2203–2209, 2000.

[40] Chen, I.-M. and Burdick, J. W., "Enumerating the non-isomorphic assembly configurations of modular robotic systems," *International Journal of Robotics Research*, vol. 17, no. 7, pp. 702–719, 1998.

[41] Chen, I.-M. and Yang, G., "Automatic model generation for modular reconfigurable robot dynamics," *Journal of Dynamic Systems, Measurement, and Control*, vol. 120, no. 3, pp. 346–352, 1998.

[42] Cheung, K. C., Demaine, E. D., Bachrach, J., and Griffith, S., "Programmable assembly with universally foldable strings (moteins).," *Robotics, IEEE Transactions on*, vol. 27, no. 4, pp. 718–729, 2011.

[43] Chiang, C.-J. and Chirikjian, G. S., "Modular robot motion planning using similarity metrics," *Autonomous Robots*, vol. 10, no. 1, pp. 91–106, 2001.

[44] Chirikjian, G. S., "Kinematics of a metamorphic robotic system," in *Robotics and Automation (ICRA), 1994 IEEE International Conference on*, pp. 449–455, 1994.

[45] Choset, H. M., *Principles of robot motion: theory, algorithms, and implementation.* MIT Press, 2005.

[46] Christensen, D. J., "Evolution of shape-changing and self-repairing control for the atron self-reconfigurable robot," in *Robotics and Automation (ICRA), 2006 IEEE International Conference on*, pp. 2539–2545, 2006.

[47] Christensen, D., "Experiments on fault-tolerant self-reconfiguration and emergent self-repair," in *Artificial Life (ALIFE), 2007 IEEE Symposium on*, pp. 355–361, 2007.

[48] Cortes, J., Martinez, S., Karatas, T., and Bullo, F., "Coverage control for mobile sensing networks," *Robotics and Automation, IEEE Transactions on*, vol. 20, no. 2, pp. 243–255, 2004.

[49] Dantu, K., Rahimi, M., Shah, H., Babel, S., Dhariwal, A., and Sukhatme, G., "Robomote: enabling mobility in sensor networks," in *Information Processing in Sensor Networks (IPSN), 2005 International Symposium on*, pp. 404–409, 2005.

[50] De, P., Raniwala, A., Krishnan, R., Tatavarthi, K., Modi, J., Syed, N. A., Sharma, S., and Chiueh, T.-c., "Mint-m: an autonomous mobile wireless experimentation platform," in *Mobile Systems, Applications and Services, 2006 International Conference on*, pp. 124–137, 2006.

[51] Donald, B. R., Levey, C. G., McGray, C. D., Paprotny, I., and Rus, D., "An untethered, electrostatic, globally controllable mems micro-robot," *Microelectromechanical Systems, Journal of*, vol. 15, no. 1, pp. 1–15, 2006.

[52] Donald, B. R., Levey, C. G., and Paprotny, I., "Planar microassembly by parallel actuation of mems microrobots," *Microelectromechanical Systems, Journal of*, vol. 17, no. 4, pp. 789–808, 2008.

[53] Dong, B. and Li, Y., "Multi-objective-based configuration generation and optimization for reconfigurable modular robot," in *Information Science and Technology (ICIST), 2011 IEEE International Conference on*, pp. 1006–1010, 2011.

[54] Dutta, A., Dasgupta, P., Baca, J., and Nelson, C., "A block partitioning algorithm for modular robot reconfiguration under uncertainty," in *Mobile Robots (ECMR), 2013 IEEE European Conference on*, pp. 255–260, 2013.

[55] Dutta, A., Dasgupta, P., Baca, J., and Nelson, C., "A fast coalition structure search algorithm for modular robot reconfiguration planning under uncertainty," in *Distributed Autonomous Robotic Systems*, pp. 177–191, Springer, 2014.

[56] 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.

[57] Fahmy, H. and Blostein, D., "A survey of graph grammars: theory and applications," in *Pattern Recognition Methodology and Systems, 1992 IAPR International Conference on*, pp. 294 –298, 1992.

[58] Feynman, R. P., "There's plenty of room at the bottom," *Engineering and Science*, vol. 23, no. 5, pp. 22–36, 1960.

[59] Fitch, R. and Butler, Z., "Million module march: Scalable locomotion for large self-reconguring robots," in *Robotics and Automation (ICRA), 2007 IEEE International Conference on*, pp. 2248 –2253, 2007.

[60] Fitch, R. and Butler, Z., "Million module march: Scalable locomotion for large self-reconfiguring robots," *International Journal of Robotics Research*, vol. 27, no. 3-4, pp. 331–343, 2008.

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

[62] Fitch, R., Butler, Z., and Rus, D., "In-place distributed heterogeneous reconfiguration planning," in *Distributed Autonomous Robotic Systems 6*, pp. 159–168, Springer, 2004.

[63] Fitch, R., Butler, Z. J., and Rus, D., "Reconfiguration planning among obstacles for heterogeneous self-reconfiguring robots.," in *Robotics and Automation (ICRA), 2005 IEEE International Conference on*, pp. 117–124, 2005.

[64] Fitch, R. and McAllister, R., "Hierarchical planning for self-reconfiguring robots using module kinematics," in *Distributed Autonomous Robotic Systems (DARS), 2013 Springer International Symposium on*, pp. 477–490, Springer, 2013.

[65] Fitch, R. C., *Heterogeneous self-reconfiguring robotics.* PhD thesis, Hanover, NH, USA, 2004.

[66] Fox, M. and Shamma, J., "Probabilistic performance guarantees for distributed self-assembly," *Automatic Control, IEEE Transactions on*, vol. 60, no. 12, pp. 3180–3194, 2015.

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

[68] Fukuda, T. and Nakagawa, S., "Dynamically reconfigurable robotic system," in *Robotics and Automation (ICRA), 1988 IEEE International Conference on*, pp. 1581–1586, 1988.

[69] Garrido-Jurado, S., Munoz-Salinas, R., Madrid-Cuevas, F. J., and Marin-Jimenez, M. J., "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280 – 2292, 2014.

[70] Gilpin, K., Kotay, K., and Rus, D., "Miche: Modular shape formation by self-dissasembly," in *Robotics and Automation (ICRA), 2007 IEEE International Conference on*, pp. 2241–2247, 2007.

[71] Gilpin, K. and Rus, D., "Modular robot systems," *Robotics Automation Magazine, IEEE*, vol. 17, no. 3, pp. 38–55, 2010.

[72] Gilpin, K., Knaian, A., and Rus, D., "Robot pebbles: One centimeter modules for programmable matter through self-disassembly.," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pp. 2485–2492, 2010.

[73] Gilpin, K., Kotay, K., Rus, D., and Vasilescu, I., "Miche: Modular shape formation by self-disassembly," *International Journal of Robotics Research*, vol. 27, no. 3-4, pp. 345–372, 2008.

[74] Gilpin, K. and Rus, D., "Self-disassembling robot pebbles: New results and ideas for self-assembly of 3d structures.," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, Workshop "Modular Robots: The State of the Art", pp. 94–99, 2010.

[75] Gilpin, K. and Rus, D., "What's in the bag: A distributed approach to 3d shape duplication with modular robots.," in *Robotics: Science and Systems (RSS), 2012 Conference on*, 2012.

[76] Goldstein, S., Campbell, J., and Mowry, T., "Programmable matter," *Computer, IEEE*, vol. 38, no. 6, pp. 99–101, 2005.

[77] Goldstein, S. C. and Mowry, T. C., "Claytronics: A scalable basis for future robots," in *RoboSphere*, 2004.

[78] Golestan, K., Asadpour, M., and Moradi, H., "A new graph signature calculation method based on power centrality for modular robots," in *Distributed Autonomous Robotic Systems*, pp. 505–516, Springer, 2013.

[79] Hamann, H., Stradner, J., Schmickl, T., and Crailsheim, K., "A hormone-based controller for evolutionary multi-modular robotics: From single modules to gait learning," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1–8, 2010.

[80] Hastings, W. K., "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[81] Hossain, S., Nelson, C. A., and Dasgupta, P., "Hardware design and testing of modred: A modular self-reconfigurable robot system," in *Advances in Reconfigurable Mechanisms and Robots I*, pp. 515–523, Springer, 2012.

[82] Hou, F. and Shen, W.-M., "On the complexity of optimal reconfiguration planning for modular reconfigurable robots.," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pp. 2791–2796, 2010.

[83] Hou, F. and Shen, W.-M., "Graph-based optimal reconfiguration planning for self-reconfigurable robots," *Robotics and Autonomous Systems*, vol. 62, no. 7, pp. 1047–1059, 2014.

[84] Joergensen, M., Ostergaard, E., and Lund, H., "Modular atron: modules for a self-reconfigurable robot," in *Intelligent Robots and Systems (IROS), 2004 IEEE/RSJ International Conference on*, pp. 2068–2073, 2004.

[85] Johnson, D., Stack, T., Fish, R., Flickinger, D., Stoller, L., Ricci, R., and Lepreau, J., "Mobile emulab: A robotic wireless and sensor network testbed," in *Computer Communications (INFOCOM), 2006 IEEE International Conference on*, pp. 1–12, 2006.

[86] Jones, C. and Mataric, M. J., "From local to global behavior in intelligent self-assembly," in *Robotics and Automation (ICRA), 2003 IEEE International Conference on*, pp. 721–726, 2003.

[87] Kavraki, L., Svestka, P., claude Latombe, J., and Overmars, M., "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," in *Robotics and Automation (ICRA), 1996 IEEE International Conference on*, pp. 566–580, 1996.

[88] Kernbach, S., "Swarmrobot. org-open-hardware microrobotic project for large-scale artificial swarms," *arXiv preprint arXiv:1110.5762*, 2011.

[89] Kernbach, S., "Encoder-free odometric system for autonomous microrobots," *Mechatronics*, vol. 22, no. 6, pp. 870–880, 2012.

[90] Kernbach, S., Schmickl, T., Hamann, H., Stradner, J., Schlachter, F., Schwarzer, C. S. F., Winfield, A. F. T., and Matthias, R., "Adaptive action selection mechanisms for evolutionary multimodular robotics," in *Artificial Life (ALIFE), 2010 IEEE Symposium on*, pp. 781–788, 2010.

[91] Kernbach, S., Thenius, R., Kernbach, O., and Schmickl, T., "Re-embodiment of honeybee aggregation behavior in an artificial micro-robotic system.," *Adaptive Behaviour*, vol. 17, no. 3, pp. 237–259, 2009.

[92] Khan, Z., Balch, T., and Dellaert, F., "Mcmc-based particle filtering for tracking a variable number of interacting targets," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 11, pp. 1805–1819, 2005.

[93] Kirby, B., Campbell, J., Aksak, B., Pillai, P., Hoburg, J., Mowry, T., and Goldstein, S. C., "Catoms: Moving robots without moving parts," in *Artificial Intelligence, 2005 AAAI National Conference on*, pp. 1730 – 1731, 2005.

[94] Klavins, E., "Programmable self-assembly," *Control Systems, IEEE*, vol. 27, no. 4, pp. 43–56, 2007.

[95] Klavins, E., Ghrist, R., and Lipsky, D., "A grammatical approach to self-organizing robotic systems," *Automatic Control, IEEE Transactions on*, vol. 51, no. 6, pp. 949 – 962, 2006.

[96] Klavins, E., Burden, S., and Napp, N., "Optimal rules for programmed stochastic self–assembly," in *Robotics: Science and Systems (RSS), 2006 Conference on*, 2006.

[97] Klavins, E., Ghrist, R., and Lipsky, D., "Graph grammars for self assembling robotic systems," in *Robotics and Automation (ICRA), 2004 IEEE International Conference on*, pp. 5293–5300, 2004.

[98] Knaian, A., Cheung, K. C., Lobovsky, M. B., Oines, A. J., Schmidt-Nielsen, P., and Gershenfeld, N., "The milli-motein: A self-folding chain of programmable matter with a one centimeter module pitch.," in *Intelligent Robots and Systems (IROS) 2012 IEEE/RSJ International Conference on*, pp. 1447–1453, 2012.

[99] Kohlstedt, K. L. and Glotzer, S. C., "Self-assembly and tunable mechanics of reconfigurable colloidal crystals," *Physical Review E*, vol. 87, no. 3, p. 032305, 2013.

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

[101] Kotay, K., *Self-reconfiguring Robots: Designs, Algorithms, and Applications*. PhD thesis, Hanover, NH, USA, 2003.

[102] Kotay, K. and Rus, D., "Locomotion versatility through self-reconfiguration," *Robotics and Autonomous Systems*, vol. 26, no. 2, pp. 217–232, 1999.

[103] Kotay, K. and Rus, D., "Generic distributed assembly and repair algorithms for self-reconfiguring robots," in *Intelligent Robots and Systems (IROS), 2004 IEEE/RSJ International Conference on*, pp. 2362 – 2369, 2004.

[104] Kotay, K. and Rus, D., "Efficient locomotion for a self-reconfiguring robot," in *Robotics and Automation (ICRA), 2005 IEEE International Conference on*, pp. 2963–2969, 2005.

[105] Kotay, K. D. and Rus, D. L., "Scalable parallel algorithm for configuration planning for self-reconfiguring robots," in *Intelligent Systems and Smart Manufacturing, 2000 SPIE Conference on*, pp. 377–387, 2000.

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

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

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

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

[110] Kurokawa, H., Tomita, K., Kamimura, A., Yoshida, E., Kokaji, S., and Murata, S., "Distributed self-reconfiguration control of modular robot m-tran," in *Mechatronics and Automation, 2005 IEEE International Conference*, pp. 254–259, 2005.

[111] Lal, S. P., Yamada, K., and Endo, S., "Emergent motion characteristics of a modular robot through genetic algorithm," in *Advanced Intelligent Computing Theories and Applications. With Aspects of Artificial Intelligence*, pp. 225–234, Springer, 2008.

[112] Larkworthy, T. and Ramamoorthy, S., "An efficient algorithm for self-reconfiguration planning in a modular robot," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pp. 5139–5146, 2010.

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

[114] Lim, Y. and Shamma, J., "Robustness of stochastic stability in game theoretic learning," in *American Control Conference (ACC), 2013*, pp. 6145–6150, 2013.

[115] Liu, J. and Wu, J., *Multiagent Robotic Systems*. CRC Press, 2001.

[116] Lyder, A., Garcia, R., and Stoy, K., "Mechanical design of odin, an extendable heterogeneous deformable modular robot," in *Intelligent Robots and Systems (IROS), 2008 IEEE/RSJ International Conference on*, pp. 883–888, 2008.

[117] Marden, J. R., Arslan, G., and Shamma, J. S., "Connections between cooperative control and potential games illustrated on the consensus problem," in *European Control Conference (ECC), 2007*, pp. 4604–4611, 2007.

[118] Marden, J. R. and Shamma, J. S., "Revisiting log-linear learning: Asynchrony, completeness and payoff-based implementation.," *Games and Economic Behavior*, vol. 75, no. 2, pp. 788–808, 2012.

[119] Marden, J., Arslan, G., and Shamma, J., "Cooperative control and potential games," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 39, no. 6, pp. 1393–1407, 2009.

[120] Marden, J., Arslan, G., and Shamma, J., "Joint strategy fictitious play with inertia for potential games," *Automatic Control, IEEE Transactions on*, vol. 54, pp. 208–220, Feb 2009.

[121] McLurkin, J., Lynch, A. J., Rixner, S., Barr, T. W., Chou, A., Foster, K., and Bilstein, S., "A low-cost multi-robot system for research, teaching, and outreach," in *Distributed Autonomous Robotic Systems (DARS), 2013 International Symposium on*, pp. 597–609, Springer, 2013.

[122] McLurkin, J., Smith, J., Frankel, J., Sotkowitz, D., Blau, D., and Schmidt, B., "Speaking swarmish: Human-robot interface design for large swarms of autonomous mobile robots.," in *AAAI Spring Symposium: To Boldly Go Where No Human-Robot Team Has Gone Before*, pp. 72–75, 2006.

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

[124] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E., "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp. 1087–1092, 1953.

[125] Michael, N., Fink, J., and Kumar, V., "Experimental testbed for large multirobot teams," *Robotics Automation Magazine, IEEE*, vol. 15, no. 1, pp. 53–61, 2008.

[126] Millan, J. A., Ortiz, D., van Anders, G., and Glotzer, S. C., "Self-assembly of archimedean tilings with enthalpically and entropically patchy polygons," *ACS Nano*, vol. 8, no. 3, pp. 2918–2928, 2014.

[127] Miyashita, S., Guitron, S., Ludersdorfer, M., Sung, C. R., and Rus, D., "An untethered miniature origami robot that self-folds, walks, swims, and degrades.," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 1490–1496, 2015.

[128] Monderer, D. and Shapley, L. S., "Potential games," *Games and Economic Behavior*, vol. 14, no. 1, pp. 124–143, 1996.

[129] Moore, G., "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–116, 1965.

[130] Murata, S., Yoshida, E., Tomita, K., Kurokawa, H., Kamimura, A., and Kokaji, S., "Hardware design of modular robotic system," in *Intelligent Robots and Systems (IROS), 2000 IEEE/RSJ International Conference on*, pp. 2210–2217, 2000.

[131] Murata, S. and Kurokawa, H., "Self-reconfigurable robots," *Robotics & Automation Magazine, IEEE*, vol. 14, no. 1, pp. 71–78, 2007.

[132] Murata, S., Yoshida, E., Kurokawa, H., Tomita, K., and Kokaji, S., "Self-repairing mechanical systems," *Autonomous Robots*, vol. 10, no. 1, pp. 7–21, 2001.

[133] Nagl, M., "A tutorial and bibliographical survey on graph grammars," in *Graph-Grammars and Their Application to Computer Science and Biology*, pp. 70–126, Springer, 1979.

[134] Nagpal, R., "Programmable pattern-formation and scale independence," in *Complex Systems (ICCS), 2002 International Confernce on*, 2002.

[135] Nagpal, R., "Programmable self-assembly using biologically-inspired multiagent control," in *Autonomous Agents and Multiagent Systems (AAMAS), 2002 International Conference on*, pp. 418–425, 2002.

[136] Neubert, J., Cantwell, A. P., Constantin, S., Kalontarov, M., Erickson, D., and Lipson, H., "A robotic module for stochastic fluidic assembly of 3d self-reconfiguring structures.," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pp. 2479–2484, 2010.

[137] Olson, E., "AprilTag: A robust and flexible visual fiducial system," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 3400–3407, 2011.

[138] Oung, R. and D'Andrea, R., "The distributed flight array," *Mechatronics*, vol. 21, no. 6, pp. 908–917, 2011.

[139] Ozgur, A., Bonardi, S., Vespignani, M., Mockel, R., and Ijspeert, A., "Natural user interface for roombots," in *Robot and Human Interactive Communication (RO-MAN), 2014 IEEE International Symposium on*, pp. 12–17, 2014.

[140] Pamecha, A., Ebert-Uphoff, I., and Chirikjian, G. S., "Useful metrics for modular robot motion planning," *Robotics and Automation, IEEE Transactions on*, vol. 13, no. 4, pp. 531–545, 1997.

[141] Park, M. and Yim, M., "Distributed control and communication fault tolerance for the ckbot," in *Reconfigurable Mechanisms and Robots (ReMAR), 2009 ASME/IFToMM International Conference on*, pp. 682–688, 2009.

[142] Parker, L. E. and Howard, A., "Experiments with a large heterogeneous mobile robot team: Exploration, mapping, deployment and detection," *International Journal of Robotics Research*, vol. 25, pp. 431–447, 2006.

[143] Petersen, K., Nagpal, R., and Werfel, J., "Termes: An autonomous robotic system for three-dimensional collective construction.," in *Robotics: Science and Systems (RSS), 2011 Conference on*, 2011.

[144] Pickem, D., Lee, M., and Egerstedt, M., "The GRITSBot in its natural habitat - a multi-robot testbed," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 4062–4067, 2015.

[145] Pickem, D. and Egerstedt, M., "Self-reconfiguration using graph grammars for modular robotics," in *Analysis and Design of Hybrid Systems (ADHS), 2012 IFAC Conference on*, 2012.

[146] Pickem, D., Egerstedt, M., and Shamma, J. S., "Complete heterogeneous self-reconfiguration: Deadlock avoidance using hole-free assemblies," in *Distributed Estimation and Control in Networked Systems (NecSys), 2013 IFAC Workshop on*, pp. 404–410, 2013.

[147] Pickem, D., Egerstedt, M., and Shamma, J. S., "A game-theoretic formulation of the homogeneous self-reconfiguration problem," in *Decision and Control (CDC), 2015 IEEE Conference on*, pp. 2829–2834, 2015.

[148] Pickem, D., Egerstedt, M., and Shamma, J. S., "Homogeneous self-reconfiguration: A game theoretic approach," *Automatic Control, IEEE Transactions on*, 2016. (submitted).

[149] Pillai, P. and Campbell, J., "Sensing and reproducing the shapes of 3d objects using claytronics," in *Embedded Networked Sensor Systems, 2006 international Conference on*, pp. 369–370, 2006.

[150] Pillai, P., Campbell, J., Kedia, G., Moudgal, S., and Sheth, K., "A 3d fax machine based on claytronics," in *Intelligent Robots and Systems (IROS), 2006 IEEE/RSJ International Conference on*, pp. 4728–4735, 2006.

[151] Pitzer, B., Osentoski, S., Jay, G., Crick, C., and Jenkins, O., "Pr2 remote lab: An environment for remote development and experimentation," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pp. 3200–3205, 2012.

[152] Ramaekers, Z., Dasgupta, R., Ufimtsev, V., Hossain, S. G. M., and Nelson, C., "Self-reconfiguration in modular robots using coalition games with uncertainty.," in *Automated Action Planning for Autonomous Mobile Robots, 2011 AAAI Workshop on*, 2011.

[153] Ray, D., *A game-theoretic perspective on coalition formation*. Oxford University Press, 2007.

[154] Romanishin, J., Gilpin, K., and Rus, D., "M-blocks: Momentum-driven, magnetic modular robots.," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pp. 4288–4295, 2013.

[155] Romanishin, J., Gilpin, K., Claici, S., and Rus, D., "3d m-blocks: Self-reconfiguring robots capable of locomotion via pivoting in three dimensions," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 1925–1932, 2015.

[156] Rowe, A., Rosenberg, C., and Nourbakhsh, I., "CMUcam: a low-overhead vision system," in *Intelligent Robots and Systems (IROS), 2002 IEEE/RSJ International Conference on*, 2002.

[157] Rubenstein, M., Ahler, C., and Nagpal, R., "Kilobot: A low cost scalable robot system for collective behaviors," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pp. 3293–3298, 2012.

[158] Rubenstein, M., Cabrera, A., Werfel, J., Habibi, G., McLurkin, J., and Nagpal, R., "Collective transport of complex objects by simple robots: theory and

experiments," in *Autonomous Agents and Multiagent Systems (AAMAS), 2013 International Conference on*, pp. 47–54, 2013.

[159] Rubenstein, M. and Shen, W.-M., "Scalable self-assembly and self-repair in a collective of robots.," in *Intelligent Robots and Systems (IROS), 2009 IEEE/RSJ International Conference on*, pp. 1484–1489, 2009.

[160] Rus, D. and Vona, M., "Crystalline robots: Self-reconfiguration with compressible unit modules," *Autonomous Robots*, vol. 10, no. 1, pp. 107–124, 2001.

[161] Russell, S. J. and Norvig, P., *Artificial Intelligence: A Modern Approach*. Pearson Education, 2010.

[162] Sadjadi, H., Al-Jarrah, M. A., and Assaleh, K., "Morphology for planar hexagonal modular self-reconfigurable robotic systems," in *Mechatronics and its Applications (ISMA), 2009 International Symposium on*, pp. 1–6, 2009.

[163] Salemi, B., Moll, M., and Shen, W.-M., "Superbot: A deployable, multifunctional, and modular self-reconfigurable robotic system.," in *Intelligent Robots and Systems (IROS), 2006 IEEE/RSJ International Conference on*, pp. 3636–3641, 2006.

[164] Schultz, U. P., Bordignon, M., and Støy, K., "Robust and reversible self-reconfiguration," in *Intelligent Robots and Systems (IROS), 2009 IEEE/RSJ International Conference on*, pp. 5287–5294, 2009.

[165] Shen, W.-M., "Self-reconfigurable robots for adaptive and multifunctional tasks," in *Autonomous/Unmanned Systems, 2008 Army Science Conference on*, 2008.

[166] Shen, W.-M., Chiu, H., Rubenstein, M., and Salemi, B., "Rolling and climbing by the multifunctional superbot reconfigurable robotic system," in *Space Technology and Applications (STAIF), 2008 AIP International Forum on*, pp. 839–848, 2008.

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

[168] Shen, W.-M., Salemi, B., and Will, P., "Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots," *Robotics and Automation, IEEE Transactions on*, vol. 18, pp. 700–712, Oct 2002.

[169] Shen, W.-M. and Will, P., "Docking in self-reconfigurable robots," in *Intelligent Robots and Systems (IROS), 2001 IEEE/RSJ International Conference on*, pp. 1049–1054, 2001.

[170] Shen, W.-M., Will, P., Galstyan, A., and Chuong, C.-M., "Hormone-inspired self-organization and distributed control of robotic swarms," *Autonomous Robots*, vol. 17, no. 1, pp. 93–105, 2004.

[171] Shiba, S., Uchida, M., Nozawa, A., Asano, H., Onogaki, H., Mizuno, T., Ide, H., and Yokoyama, S., "Autonomous reconfiguration of robot shape by using q-learning," *Artificial Life and Robotics*, vol. 14, no. 2, pp. 213–218, 2009.

[172] Siciliano, B. and Khatib, O., *Springer handbook of robotics*. Springer Science & Business Media, 2008.

[173] Sproewitz, A., Pouya, S., Bonardi, S., van den Kieboom, J., Moeckel, R., Billard, A., Dillenbourg, P., and Ijspeert, A., "Roombots: Reconfigurable robots for adaptive furniture," *Computational Intelligence Magazine, IEEE*, vol. 5, pp. 20–32, Aug 2010.

[174] Sproewitz, A., Billard, A., Dillenbourg, P., and Ijspeert, A. J., "Roombots-mechanical design of self-reconfiguring modular robots for adaptive furniture.," in *Robotics and Automation (ICRA), 2009 IEEE International Conference on*, pp. 4259–4264, 2009.

[175] Sproewitz, A., Moeckel, R., Vespignani, M., Bonardi, S., and Ijspeert, A. J., "Roombots: A hardware perspective on 3d self-reconfiguration and locomotion with a homogeneous modular robot," *Robotics and Autonomous Systems*, vol. 62, no. 7, pp. 1016 – 1033, 2014.

[176] Stoy, K. and Nagpal, R., "Self-reconfiguration using directed growth," in *Distributed Autonomous Robotic Systems 6*, pp. 3–12, Springer, 2007.

[177] 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, no. 4, pp. 410 –417, 2002.

[178] Støy, K., "Controlling self-reconfiguration using cellular automata and gradients," in *Intelligent Autonomous Systems (IAS), 2004 International Conference on*, pp. 693–702, 2004.

[179] Stoy, K., "How to construct dense objects with self-recondfigurable robots," in *European Robotics Symposium 2006*, pp. 27–37, 2006.

[180] Stoy, K., "Using cellular automata and gradients to control self-reconfiguration," *Robotics and Autonomous Systems*, vol. 54, no. 2, pp. 135–141, 2006.

[181] Stoy, K., Brandt, D., and Christensen, D. J., *Self-reconfigurable robots: an introduction*. MIT Press, 2010.

[182] Stoy, K. and Nagpal, R., "Self-repair through scale independent self-reconfiguration," in *Intelligent Robots and Systems (IROS), 2004 IEEE/RSJ International Conference on*, pp. 2062–2067, 2004.

[183] Stradner, J., Hamann, H., Schmickl, T., Thenius, R., and Crailsheim, K., "Evolving a novel bio-inspired controller in reconfigurable robots," in *Advances in Artificial Life (ECAL), 2009 European Conference on*, vol. 5777 of *Lecture Notes in Computer Science*, pp. 132–139, Springer, 2011.

[184] Sung, C., Bern, J., Romanishin, J., and Rus, D., "Reconfiguration planning for pivoting cube modular robots," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 1933–1940, May 2015.

[185] Thrun, S. and Liu, Y., "Multi-robot slam with sparse extended information filers," in *Robotics Research, 2005 International Symposium on*, pp. 254–266, 2005.

[186] Tolley, M. T. and Lipson, H., "On-line assembly planning for stochastically reconfigurable systems.," *International Journal of Robotics Research*, vol. 30, no. 13, pp. 1566–1584, 2011.

[187] Tolley, M. T. and Lipson, H., "Programmable 3d stochastic fluidic assembly of cm-scale modules.," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pp. 4366–4371, 2011.

[188] van Anders, G., Ahmed, N. K., Smith, R., Engel, M., and Glotzer, S. C., "Entropically patchy particles: Engineering valence through shape entropy," *ACS Nano*, vol. 8, no. 1, pp. 931–940, 2013.

[189] Vassilvitskii, S., Yim, M., and Suh, J., "A complete, local and parallel reconfiguration algorithm for cube style modular robots," in *Robotics and Automation (ICRA), 2002 IEEE International Conference on*, pp. 117 – 122, 2002.

[190] Walter, J. E., Tsai, E. M., and Amato, N. M., "Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots," *Robotics, IEEE Transactions on*, vol. 21, no. 4, pp. 621–631, 2005.

[191] Wei, H., Li, H., Tan, J., and Wang, T., "Self-assembly control and experiments in swarm modular robots," *Science China Technological Sciences*, vol. 55, no. 4, pp. 1118–1131, 2012.

[192] White, P., Zykov, V., Bongard, J. C., and Lipson, H., "Three dimensional stochastic reconfiguration of modular robots.," in *Robotics: Science and Systems (RSS), 2005 Conference on*, pp. 161–168, 2005.

[193] Whitesides, G. M. and Boncheva, M., "Beyond molecules: Self-assembly of mesoscopic and macroscopic components," *Proceedings of the National Academy of Sciences*, vol. 99, no. 8, pp. 4769–4774, 2002.

[194] Xue, Y. and Grover, M., "Optimal design for active self-assembly system," in *American Control Conference (ACC), 2011*, pp. 3269–3274, 2011.

[195] Yim, M., Roufas, K., Duff, D., Zhang, Y., Eldershaw, C., and Homans, S., "Modular reconfigurable robots in space applications," *Autonomous Robots*, vol. 14, no. 2-3, pp. 225–237, 2003.

[196] 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," *Robotics & Automation Magazine, IEEE*, vol. 14, no. 1, pp. 43–52, 2007.

[197] Young, H. P., "The evolution of conventions," *Econometrica: Journal of the Econometric Society*, pp. 57–84, 1993.

[198] Young, P., *Individual Strategy and Social Structure: An Evolutionary Theory of Institutions*. Princeton University Press, 1998.

[199] Yu, C.-H., Haller, K., Ingber, D., and Nagpal, R., "Morpho: A self-deformable modular robot inspired by cellular structure," in *Intelligent Robots and Systems (IROS), 2008 IEEE/RSJ International Conference on*, pp. 3571–3578, 2008.

[200] Yu, C.-H., Willems, F.-X., Ingber, D., and Nagpal, R., "Self-organization of environmentally-adaptive shapes on a modular robot," in *Intelligent Robots and Systems (IROS), 2007 IEEE/RSJ International Conference on*, pp. 2353–2360, 2007.

[201] Zhu, M. and Martinez, S., "Distributed coverage games for energy-aware mobile sensor networks," *SIAM Journal on Control and Optimization*, vol. 51, no. 1, pp. 1–27, 2013.

[202] Zykov, V., Chan, A., and Lipson, H., "Molecubes: An open-source modular robotic kit," in *Workshop on Self-Reconfigurable Robotics at Intelligent Robots and Systems (IROS), 2007 IEEE/RSJ International Conference on*, 2007.

[203] Zykov, V., Mytilinaios, E., Desnoyer, M., and Lipson, H., "Evolved and designed self-reproducing modular robotics," *Robotics, IEEE Transactions on*, vol. 23, no. 2, pp. 308–319, 2007.