AUTONOMOUS RECONFIGURATION PLANNING IN MODULAR ROBOTS

By

Rowan McAllister

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF ENGINEERING
AT
UNIVERSITY OF SYDNEY
SYDNEY, NEW SOUTH WALES
5 NOVEMBER 2009

DECLARATION

Author: **Rowan McAllister** Date: **5 November 2009**

Title: **Autonomous Reconfiguration Planning in Modular Robots**

Department: **Engineering and Information Technologies**

Degree: **B.E.** Convocation: **November** Year: **2009**

Declaration The author attests that permissions has been obtained for the use of any copyrighted material appearing in this thesis (other than brief excepts requiring only proper acknowledgement in scholarly writing) and that all such use is clearly acknowledged. This is all entirely my own work unless explicitly referenced. Specific tasks conducted in this thesis include:

- Literature review of modular robotic hardware, history and design procedures
- Literature review and of self reconfiguration planners, drawing from the many available papers to develop a survey of planners
- Development of module representations and all program routines from both planners with the exception of *Connectivity Checker* which was borrowed and edited from the Million Module March algorithm from Fitch and Butler
- Analysis of both designs
- Conducting all reconfiguration simulations and statistics compiled to verify design goals. Simulations used code from the *Superbot Simulator* written by David Brandt from the ISI
- Helping to assemble the Hardware-in-the-Loop simulator
- Recommendations for final testing procedures, and future work proposed

Author: ____________________________
Rowan McAllister

Supervisor: ____________________________
Dr. Robert Fitch

To Mum, Dad, Marion and Emily

Table of Contents

Abstract

Self-Reconfiguring Robots (SRR) are composed of many modules that have the ability to autonomously attach and detach, enabling adaptation to a variety of tasks in unknown surroundings. In order to change shape into a particular form, an SRR must plan a sequence of module movements. This *reconfiguration problem* is challenging because of the many mechanical degrees of freedom and the resultant large number of possible SRR configurations which contribute to a vast and high-dimensional search space. To support the operation of an SRR in a practical environment, reconfiguration planners must satisfy a number of properties including decentralized computation, parallel motion of modules, real-time execution and planning in the native kinematic space of the module mechanism. The ultimate solution would be a general planner that solves reconfigurations of arbitrary module designs in this way. This thesis has taken a step in this direction of generality by developing a reconfiguration planner of the 3R module that is easily instantiable to other module types. The planner is scalable and decentralized, and considers the native kinematics of a module. It demonstrates the ability to coordinate the parallel motion of modules and executes in real-time. The thesis presents both centralized and decentralized implementations of the algorithm along with performance evaluation for several reconfiguration examples, as well as an analysis of achievable SRR reconfigurations.

Acknowledgments

I would like to thank Robert Fitch, my supervisor, for his many suggestions and constant support during this research.

I am also thankful to my fellow honors students: Ben Itzstein, Brandon Navra and Michael West, for their valuable feedback during the seminar period and help on many technical problems. I had the pleasure of meeting Ritesh Lal, another member of our modular robotics team, whose expertise in the hardware components enables myself to generate real world results in testing. This thesis also made use of the Superbot simulator written by David Brandt under the auspices of ISI.

Of course, I am grateful to my parents for their patience and *love*. Without them this work would never have come into existence.

Finally, I wish to thank the following: Hugh for his interest, encouragement and feedback, Andrew (for being there, and ironically enough; keeping me sane), Monica and Stace (for all the coffees and being great people to live with), Behny for your patience, John for your enthusiastic help in editing, Davo, Taro, Kay, Sandy, Daniel, Tim, Stirling, Alice, Yannick, Ang, Jenny, Chris, Ben, Evan, Jill, Prabhat, Paul, Mick, Mugs & Mike.

Sydney, New South Wales   Rowan McAllister

November 5, 2009

Preface: Team Context

This thesis is part of a team effort at the ACFR. There are three other undergraduate honors students involved in this project on modular robots. Their thesis focuses are:

Ben ITZSTEIN: ***Stability, Reconfiguration Planning***

Reconfiguration planning of generalized modules with attention on a robot's static stability. In contrast this thesis planner plans in native kinematic space, using the specific hardware design of a non-generalized module to reconfigure.

Brandon NAVRA: ***Localization***

An investigation of robot localization using ultra-wide band (UWB) radar technologies to sense changes in an environment as well as establish communication networks between other robots. This focuses on using UWB radars to localize a team of modular robots.

Michael WEST: ***Stability***

A focus on static stability during reconfiguration, a challenging open problem as modules move asynchronously and in parallel. It presents a stability checking algorithm based on dynamically fusing position data from each module to obtain estimates of the mass center and the convex hull of the robot's footprint.

List of Acronyms

ACFR Australian Center for Field Robotics

ACW Anticlockwise

BFS Breadth First Search

CSRP Centralized Self Reconfiguring Planner

CW Clockwise

DFS Depth First Search

DOF Degrees Of Freedom

DSRP Decentralized Self Reconfiguring Planner

EPFL Ecole Polytechnique Fédérale de Lausanne (Swiss Federal Institute of Technology in Lausanne)

GUI Graphic User Interface

IDFS Iterative Depth First Search

LSN Local State Network

MMM Million Module March

RL Reinforcement Learning

SD Secure Digital memory card

SMA Shape Memory Alloy

SRP Self Reconfiguration Program

SRR Self Reconfiguring Robot

UWB Ultra Wide Band radar

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

"The one who adapts his policy to the times prospers, and likewise that the one whose policy clashes with the demands of the times does not" - Niccolo Machiavelli

Adaptability has always been a key trait to continued success. If any entity can either alter itself or its methods to better suit new conditions in its environment, it will either live on or experience continued use. Modular robotics has been a significant step in this direction for robotic design. In contrast to conventional robotic designs, which use fixed morphologies and have a limited repertoire of performable tasks, modular robots can adapt and assume different shapes to better suit their environment, new tasks or when they need to change location.

Conceptually, modular robots are similar to Lego® where one has many similar parts which can clip together to form a particular structure. Furthermore many different structures are possible, such as those shown by Fig. 1.1 of a 4-legged walker, cylinder or tank tread.

A key advantage of modular robotics is that their configuration is never permanent.

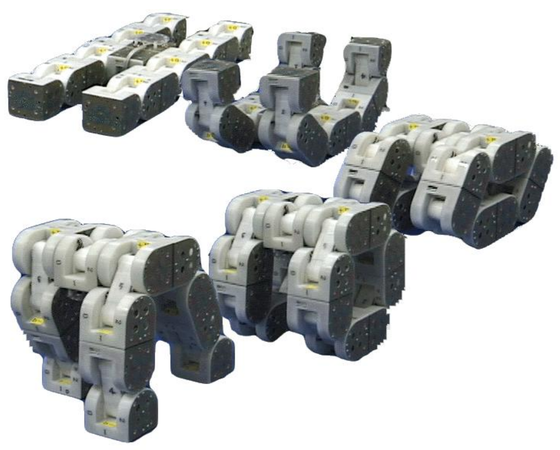


Figure 1.1: A MTRAN based robot in various configurations, from AIST [20]

They can always be pulled apart into their separate modules, and these modules can be used again to build a completely new robot. A Self Reconfiguring Robot (SRR) is a modular robot which changes its own configuration autonomously. Such ability could prove very useful for a robot in the field far from external help.

An SRR reconfiguration poses a computational problem: the robot must have a method of instructing each module about how it should move and at what time to reconfigure correctly. Although there are many different video examples of SRRs demonstrating change of shape, these are the result of manually coded algorithms which have been coded for specifically predetermined action sequences. Gait tables have also been used. Gait tables map a limited set of situations and designate a specific set of action responses for the robot which fit with those particular situations. Some of these gait tables, such as Polybots pedaling of a childs tricycle [51, p.33] have demonstrated some impressive results. However, to fully achieve true autonomous reconfigurations, an SRR requires a Self Reconfiguration Program (SRP). Currently several SRPs exist with one of two shortcomings; they are purpose-built and limited to use on a particular module design, or they are overly generic so as to encompass many module designs at the gross expense of reconfiguration efficiency. Our vision is a general planner that is completely ‘kinematically aware’ of arbitrary module designs inputted to act a ’purpose-built’ SRP for SRRs of all module-types. This combines the advantages of encompass many module designs with the reconfiguration efficiencies achievable by purpose-built SRPs. This is the fundamental algorithmic challenge in the entire field of SRRs. The purpose of this thesis is to help progress the field towards generality.

1.1 Motivation

The development of a fully autonomous SRP can significantly improve the overall autonomy of a modular robot, enhancing the robot’s efficiency and utility. The motivation for this lies in the new and unique benefits that modular robotic systems can offer society. Modular robots have several key aspects of their inherent design that makes them extremely adaptable, robust and potentially cost effective. These strengths lead to several interesting future applications, particularly in space exploration and disaster relief.

1.1.1 Benefits of Modular Robots

Adaptability

This can be seen if such a robot were to transverse a landscape of varying terrains. For flat ground, a self-propelling tank tread configuration (Fig. 1.1, *rightmost configuration*) could be used to travel quickly. However as the terrain becomes increasingly uneven, a multi-legged robot configuration would be more adept at climbing over rocks or straddling ditches. Additionally snake-like configurations have shown to be very capable at swimming and crawling through small gaps if the robot came across a river or thick scrub [6, 48].

Fault Tolerance

SRRs have a high degree of redundancy. In a *homogeneous* SRR, every part of the robot is the same as every other part, so there is never one part of the robot critical to its overall operation. If the robot is damaged in a certain area, it can simply discard the broken modules and replace them with any other modules in the system, and continue on as normal. This method of self-repair still enables the SRR to achieve close to previous performance, especially if it had many modules to begin with, allowing for graceful degradation of the system. It makes for an extremely robust robot design, with a large potential for recovery.

Cost Effectiveness

SRRs have the potential to be made at low cost if the demand for them is high enough. Being comprised of similar modules, the mass production of these modules is entirely realistic.

1.1.2 Applications

Modular robots could never be expected to work as fast or as precise as a factory robot designed for a particular task in a controlled environment, but their value is displayed in the ability to perform a wide variety of tasks, many perhaps unanticipated by initial designers, in uncontrolled environments. The current state of the modular robotics field is still in its infancy, however with further development several applications have been proposed [49]:

Space Exploration

Autonomous machines in space exploration are confronted with a variety of special challenges. Firstly, all cargo transported via shuttle is subject to strict dimensional constraints

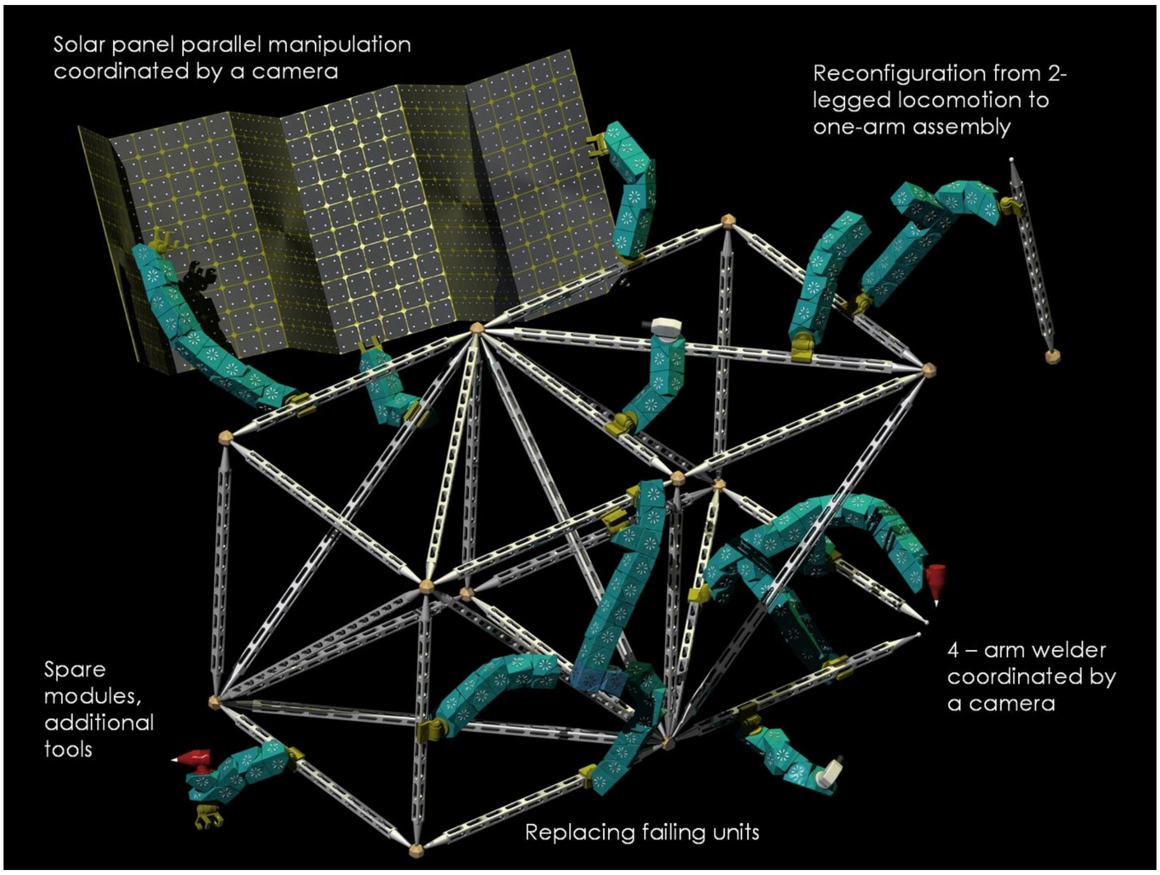

Solar panel parallel manipulation coordinated by a camera
Reconfiguration from 2-legged locomotion to one-arm assembly
Spare modules, additional tools
4 – arm welder coordinated by a camera
Replacing failing units

Figure 1.2: An artist's depiction of chain-type modular robots cooperatively assembling a truss structure in space, from Zykov *et al.* [55]

for packing reasons. Since modular robots can assume arbitrary shapes they could be expected to meet these constraints as long as they do not exceed them by volume. Additionally if the robot was the last object to pack, its packing flexibility could be taken advantage of by filling some irregular cavity in the cargo assortment that no other object could be expected to fill.

Secondly many unmanned exploratory space missions are one-way [7], where a vessel must make do with any mishaps it encounters without hope of external repair or maintenance. Hence robustness and the ability of self repair counts for a lot, but adaptability can certainly help a robot from getting into trouble in the first place. For example both the *Spirit* and *Opportunity* Mars Rover missions nearly met with disaster when hit by a severe dust storm in June of 2007 [30] . Both rovers were covered in layers of dust, blocking much of the sun's light reaching their solar panels. *Spirit's* power generation was reduced to just 18.3% of normal supply. Another dust storm in November 2008 further reduced this level to 12.7% which was just above the critical levels of power needed to run internal heaters protecting scientific equipment onboard [31]. Unfortunately even though both storms were predicted the rovers could not do anything to protect their panels from the dust. In this case a modular robotics design could be at an advantage being able to form a protective shell around any such solar panels and waiting the storm out.

Additionally the designers of the Martian Rovers were able to take advantage of a wealth of knowledge of Mars' terrain from the many previous probes sent to Mars in developing the Rover design [7]. First-time missions to new planets or moons could not expect the same luxury. An interesting mission would arguably be one to the geologically active moon Titan orbiting Saturn. In 2004 the *CassiniHuygens* mission discovered the presence of liquid hydrocarbon lakes over the surface of Titan. In addition to weather conditions of wind and rain which fluctuate seasonally, the moon has a variety of landscapes including ice, sand dunes and rocky outcrops [43]. If a '*Titan-Rover*' was developed similarly to the *Spirit* or *Opportunity* and landed over one of these lakes, it may not fair so well, however modular robots (once configured correctly) have shown to be very capable swimmers [48]. In addition a modular robot would not be as bounded by the different terrains or as susceptible to varying weather conditions as a fixed-design Rover would. It could thus be expected to travel to a variety of locations and discover more about Titan's dynamic geology than a Rover.

Even at low to medium earth orbits where it is possible to send technicians to repair or modify equipment such as satellites, the Hubble space telescope or the International Space Station, the financial cost of doing so is considerable, where the cost of launching a shuttle launch is US$450 million on average (2008 dollars) [32]. So robustness of design is still paramount in any region of space, which is why using modular robots could be a significant advantage to future space missions.

Disaster Relief

Modular robots could also potentially be used in instances of disasters, particularly collapsed buildings. Often it is difficult to locate persons trapped under much ruble and even if they are it can be very hard to aid them until excavators have removed obstructing ruble. In cases like these, modular robots have the ability to be fed single file through a small gap and later morph into something more useful once inside a cavity where a survivor lies. From here it could possibly provide temporary medical assistance, help move debris the survivor is trapped under or even serve as a communications channel between rescue workers and the survivor.

Bucket of Stuff

Lastly a *bucket of Stuff* has been proposed by many modular robotics groups. This is a general idea that in the future, a regular person may store a bucket of modules in their garage and use it as a general-applications robot for mundane tasks like cleaning gutters, feeding the dogs, guarding the house etc. It could also be used to form any static object for temporary use such as a chair, coffee table or ladder as required.

1.2 Problem Statement

To realize these applications a reliable SRP needs to be implemented such that an SRR can at least reconfigure between configurations intended for its use. Ideally an SRP should be able to reconfigure between arbitrary configurations. To achieve this, this thesis planner is:

- **Decentralized & parallel:** operates over a distributed network of module microchips planning the parallel motion of modules
- **Executes in real-time**
- **Kinematically aware:** considers a module's native kinematics; all actions possible of a given module design
- **Scalable:** requires a minimum of module movements such that reconfigurations times are fast and energy is conserved
- **General:** is written as hardware-dynamic as time allows, ensuring that most, if not all planner software is portable between robots of differing module types

The completion of this thesis planner will serve two purposes:

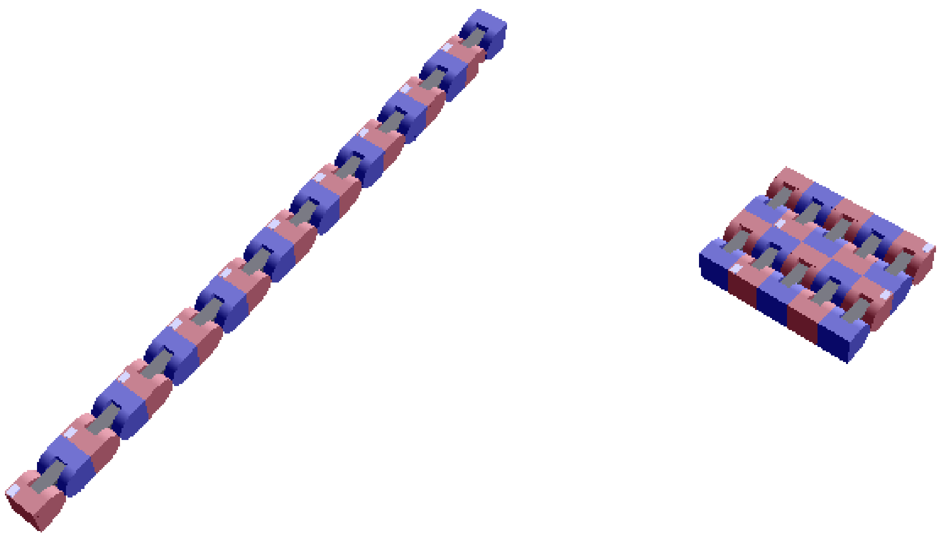
1. A better scientific understanding of planning in SRRs

2. Use as a design optimization tool for the ACRF's module. Since 1988 there have been more than 25 complete and working designs of these modular units developed around the world [33] [50, p.48]. At the Australian Center for Field Robotics (ACFR) within the University of Sydney, there is a team of 11 [1] working towards developing a unique and independent module design of our own. At the present time there is no theoretical underpinning that can compare module designs to determine which would offer an SRR the most flexibility and efficiency in its reconfiguration options [33, p.166]. However with a generalized planning algorithm that can plan SRR reconfigurations for arbitrary module designs, a comprehensive range of designs can be tested in simulations of reconfigurations in order to determine the 'optimum' module design

1.2.1 Ultimate Goal

An ultimate goal of this thesis is: '*To create a decentralized program that can autonomously plan arbitrary & parallel reconfigurations of homogeneous modular robots of arbitrary module designs in native kinematic space*'. The exact meaning of each term is explained progressively throughout the next few chapters. This goal is an ambitious one and is not expected to be fully achieved however serves to continually direct the progress of this thesis' SRP.

[1]At the time of this writing (9 Sept. 2009); 4 honors students, 1 PhD student, 2 engineering and a staff supervisor, 3 international collaborators (EPFL Switzerland, University of Southern Denmark, Zach Butler RIT) and a local collaborator



(a) state-A (b) state-B

Figure 1.3: Two possible configurations of a Modular Robot

1.3 Challenges

A major difficulty in reconfiguration planning lies in the myriad of unique configurations an SRR can be in. As an example; the configuration labeled *state-A* is one possible configuration of a 10-module robot, and *State-B* is another. In order to change from *State-A* into *State-B*, there is no one atomic action the robot can perform to do this, instead it will have to pass through a host of interim configurations, as modules move about, to slowly extrude out into the *State-B* shape.

To help visualize this reconfiguration task, of transferring from *State-A* into *State-B*, let every unique configuration the robot can be in be represented as a single node (circle) in Fig. 1.4. *State-A* will be one of these unique configurations, and *State-B* will be another. The black lines that link these nodes are indicative of actions which the robot can take, such as one or more modules moving. Every time a module moves, the robot's overall configuration

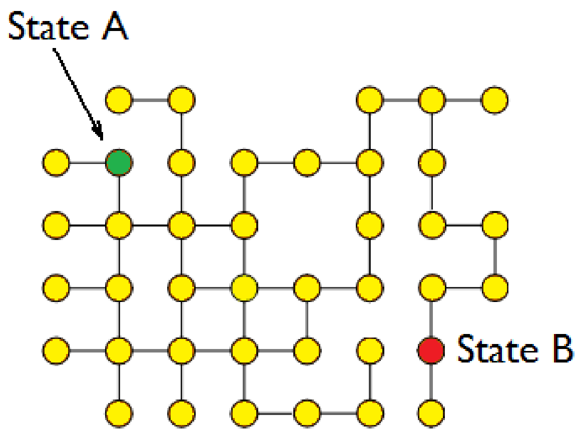


Figure 1.4: State-Network Visualization

is inevitably altered, which is represented here as a neighboring node; a transition to a different configuration. This forms a searchable network of configurations or *states*, where the nodes represent robot's unique configurations, and the lines that link them represent atomic actions the robot can take (module movements).

However this state-network grows exponentially with the number of modules the robot is made up of. With ten 3-R type modules (Fig. 1.3), there are over 3.5 trillion trillion possible configurations this robot can be in, which becomes the size of the state-network that needs to be searched, a formidable search task. At just 33 modules, this number exceeds the number of atoms in our known universe[2], so even a heuristic-based search like A-Star cannot solve this problem.

1.4 Approach

To solve this problem of a massive searchable state space, this thesis has looked to hierarchical planning. This helps divide up the main goal of finding a path between two

[2]approx. 10^{78} to 10^{82} [10, p.37]

configurations into a set of sub-goals that normal search routines can handle. If subgoals can be placed at close intervals along the path from the present configuration to desired one, regular search routines can be expected to link them up. And by connecting the dots, a solution path is found.

A hierarchical search algorithm that has inspired the planning algorithm of this thesis is Million Module March (MMM) by Fitch and Butler (2008) [14]. It is a general planning algorithm that addresses the locomotion of SRRs made from many modules, represented as sliding cubes. Beginning with an initial robot configuration it is able to coordinate many surface modules to move simultaneously and converge into a given goal-shape without relying on a global synchronization of movements. Additionally the time the algorithm takes to execute planning is sub-linearly proportional to the number of modules (cubes) present which makes for a very scalable program, enabling a single processor to plan for robots of thousands or even millions of modules.

The MMM is similar in intent to the ultimate goal of this thesis; to develop a generalized reconfiguration planner for SRRs of arbitrary modules design. However where the MMM plans for generalized cubes, this thesis goal plans for modules of defined designs, taking into account their structure and native kinematics and hence its implementation is very different.

1.4.1 Assumptions

The assumptions made in developing this reconfiguration planner are:

- Friction between module movements can always be overcome by the module's actuators

- Module locations keep to their lattice (discrete) positions in 3D space
- Adjacent connectors are always aligned, and thus able to perform connection/disconnection actions

To account for forces such as gravity and inertia that will inevitably effect module movements, the planner makes a conservative assumption that module actuators have sufficient torque to move a single module not including itself. A module's actuators may well have the ability to move several modules against gravity, this depends entirely on the hardware design. However by limiting the planner to single module movements, it can expect that such movements can be preformed wherever and whenever required without fail. This is a reasonable assumption because modules with only enough torque to reorient themselves (and not other modules) fall prey to robot configurations which are impossible to reconfigure, and a robot composed of such modules would be of little use.

1.5 Summary of Results

This thesis has resulted in a new reconfiguration planner (DSRP) that can solve arbitrary robot reconfigurations of the 3R module. The DSRP is not fully general but can easily be instantiated to other module types by similar approach because all high level DSRP routines are general. This design is the first general purpose reconfiguration planner for the Superbot module. It is also the first decentralized, real-time, non-metamodule planner of arbitrary goals shapes that can additionally direct the parallel motion of modules. The DSRP is efficient; its execution time is a constant time factor over that of the Million Module March (MMM) algorithm which is the most scalable algorithm that exists for planning[3]. This allows the design to plan for robots of many modules and still run in real time.

[3]the MMM algorithm has reconfigured 2.2 million modules in simulation [14]

1.6 Thesis Outline

This report proceeds with an introduction to SRR hardware in chapter 2 followed by an overview of reconfiguration planning in chapter 3 including a survey of exiting SPRs[4] and review of reinforcement learning. Chapters 4 and 5 outline the approach this thesis took to develop two planners, a centralized planner that moves modules in serial, and a decentralized implementation that is able to move modules in parallel. Both chapters conclude with an analysis and discussion of planner performance. Chapter 6 draws conclusions from this thesis and discusses future work. The terminology of Self Reconfiguring Programs can often appear esoteric, and so a glossary is located at the back for quick reference.

[4]Several surveys of hardware systems have already been published in the literature [23, 33], however no such comparative survey has yet been published of Self Reconfiguration Programs even though many now exist

Chapter 2

Background: SRR Hardware

Before progressing into the design aspects of this thesis' reconfiguration planner, it is necessary to review components of current literature that share common application. This chapter begins with an introduction to the field of modular robots for the purpose of setting all subsequent discussion into context. This review then progresses with an analysis of existing module hardware, discussing some of the advantages, disadvantages and design choices made by past module designers. Reconfiguration planning is tightly coupled to module hardware. This is especially so when planning in native kinematics or in the design of purpose-built planners of particular module designs. Hence this chapter also serves as necessary the background for a discussion of self reconfiguration planners in chapter 3.

2.1 History

Modular robotics as a research field is still in its infancy, the first related publication presented in 1987 [16]. To shed light on the context of its development however, this section firstly summarizes a brief history of general robotics.

The first digitally operated programmable robot used in industry was *Unimate* used in a General Motors assembly line in 1961 which “Obeying step-by-step commands stored on a magnetic drum, the 4,000-pound arm sequenced and stacked hot pieces of die-cast metal” [39]. From this demonstration of how programmable machines could be used to perform certain tasks cheaper and with greater reliability and accuracy than human personnel, robots soon experienced widespread use in manufacturing industries.

By the 1970’s, industrial robot use had spread to welding, paint spraying and grinding, which required frequent replacements of the end effectors (manipulators), due to clogged nozzles, worn grinding wheels etc. This gave rise to the ‘quick change’ end-effector design, the use of standardized connection interfaces from which a manipulator (e.g. a nozzle) could be detached completely and a replacement one reattached quickly and easily [50]. The ‘quick change’ design not only allowed for rapid replacement of manipulators but also the rapid change of manipulators, so a grinding robot could become a sander or a buffer in a short amount of time.

The advent of changeable manipulators certainly expanded the capabilities of some robots, however not to a limitless degree. The majority of factory robots remain as fixed-morphology designs to this day and thus are not fully adaptable to a wide range of different tasks.

In 1987, a robotics researcher by the name of Toshio Fukuda saw great value in the quick change mechanism and voiced how this concept could be extended from changing just the end effector to being able to change any component of the robot’s hardware including arm joints, power system, the mobile mechanism or even the software in his publication titled ‘*Approach to the Dynamically Reconfigurable Robotic System*’ [16]. Here

Fukuda outlined how this concept already exists in nature, drawing similarities between organic cells and the robotic modules he was proposing. Cells can operate independently and their local functions can be quite simple, but when they cooperate and self-assemble into something bigger they can perform a vast variety of complex tasks. Organisms also have the ability to replace cells as some inevitably die. Fukuda believed robotics could be designed as such which would lead into next generation of robotic systems [17, p.1585].

Fukuda's pioneering work in this field illustrated many of the challenges concerning modular systems that are still open research topics today including [16]:

1. The communication methods between cells
2. The control method of the approach between two cells
3. The control method of the connection and detachment between two cells
4. The method for making optimal configurations depending on a given task
5. The detection method for the malfunctioning modules due to degradation
6. The control method for the restoration and the reconstruction. In case all functional requirements cannot be met, some degrading control methods must be generated as an alternative

Two of Fukuda's papers published in 1988 [17, 18] that are more commonly referenced explore solutions to the research questions he posed above, and in 1989 developed the first design of a module; the *CEBOT* [18]. Fukuda has since been recognized as the pioneer of 'modular robotics' by the modular robotics international community.

Since the field's conception there have been over 300 publications in international journals and conference events [45] and more than 25 complete and working module designs for modular robots developed [50, p.48].

2.2 Module Design

Modules are the building blocks of modular robots. There are currently many different complete and working designs of modules around the world since the field's conception in 1988, as shown by a comprehensive survey by Østergaard *et al* [33].

All module designs include at least 2 docking interfaces to form physically rigid connections with other modules. This allows modular robots to form structurally stable configurations, however these interfaces also provide means of hardwired communication and power transfer between modules. Most modules can also bend in certain places, using hinges which are controlled by a set of internal actuators. This allows modules several 'degrees of freedom' to swivel connecting faces around. These degrees of freedom amongst individual modules form the basis in which a modular robot can reconfigure.

In addition to connection interfaces, hinge joints and actuators, modules can also be outfitting with (but not limited to) the following components:

- **Power Supply:** To power the module's active elements
- **Microchip:** For autonomous control
- **Memory:** To store procedures, current state information, etc.
- **Sensors:** To aid module-module docking, or perceiving nearby obstacles

2.2.1 Types of Modular Robots

There are several sub-categories modular robots can fall under [50]. Each of the following examples have significant implications for the hardware and software design and impose respective advantages & disadvantages to a robot. The two major classification types are:

- **Architectural Classification:** Dependent on geometric arrangements of connected modules (*e.g. Chain, Lattice, Hybrid, Mobile*)
- **Compositional Classification:** The composition of module type(s) the robot is based from (*e.g. Homogeneous, Heterogeneous*)

This thesis is specifically concerned with homogeneous hybrid modules.

Chain Architectures

Chain architectures (also *Tree* or *Continuous* architectures) are modular robots of string-like topologies akin to robotics arms with joints at regular intervals. Each module joint can move continuously between arbitrary angles such that an end effector can potentially reach any point & orientation in space that does not exceed its arm length. The cost of this versatility is computational complexity of control due to an infinitum of possible configurations. Furthermore both motion and control are executed in serial only; a chain (or each branch of a *tree* topology) is limited to performing one task at a time.

Because chain architectures do not exploit lattice regularity of module positions, they are required to have some other methods of ensuring connector alignment when forming new connections. Such methods rely on active sensing which can be more error prone, and sensor based reconfiguration research is currently underdeveloped [33, p.171].

Lattice Architectures

Lattice architectures (also *Crystal* or *Discrete* architectures) are designed to pack modules together into 2D or 3D tessellations, such as regular cubic or hexagonal grid pattern, akin to molecules in a crystal. All modules exist in one of the discrete locations defined by the grid pattern except for those currently moving between locations. An advantage here in software planning is that a program only ever need consider modules at discrete locations. Module motion options are also discrete and bounded. This allows for tightly bounded search spaces in which an SRP can feasibly plan module movements in parallel.

A disadvantage of this design is the gross limitation of module placements. Their positions cannot be fine tuned for any reason such as picking up an object or perform continuous motions needed for tasks like welding. Additionally modules are never perfectly rigid and will bend when forming cantilever shapes. If the end-most module moves outside lattice tolerances, they can cause connection misalignments and module collisions. Østergaard *et al.* conducted a test of the mechanical deformation of their ATRON design (sec. 2.3.3) as seen in Fig. 2.1 exploring this effect.

Hybrid Architectures

Hybrid designs are the marriage of lattice and chain architectures combining the advantages of each. A hybrid module's exterior is as a lattice module to pack together in a grid pattern if required, but module movements are not necessarily discreet. A hybrid module could move part way such that its position (and those connected to it) does not align with a global grid pattern. This is merely an option for the robot, to use its modules as chain-modules if desired for precision tasks. This does incur the associated computations complexities discussed, however it can always resort to a purely lattice architecture robot if desired.

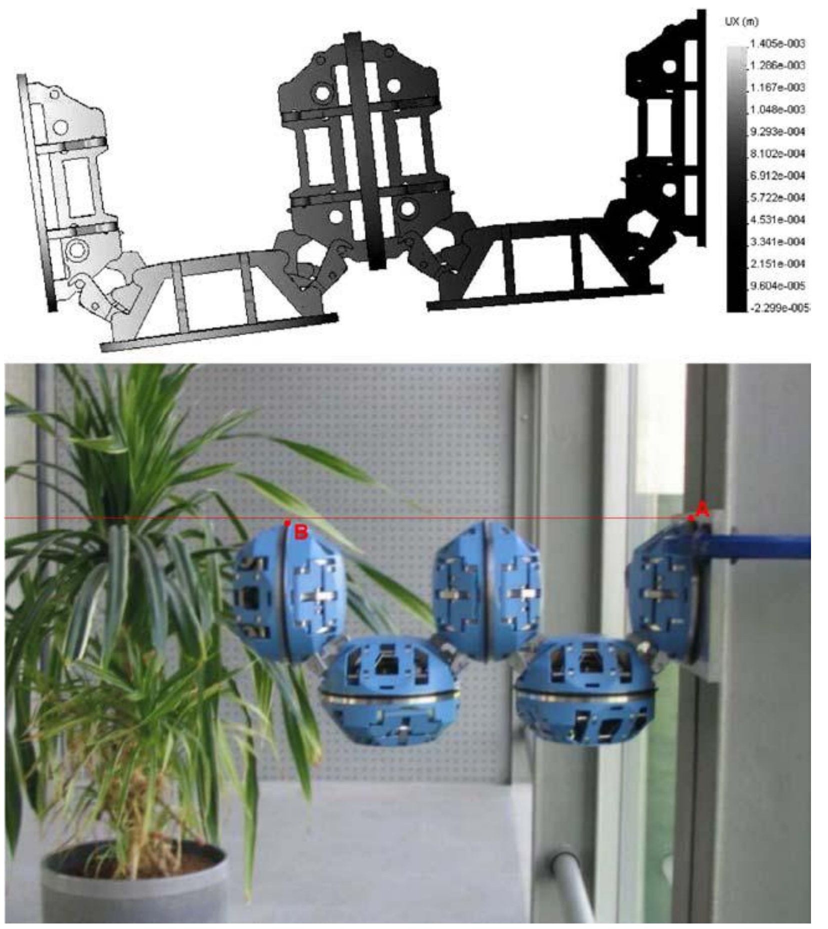


Figure 2.1: “Mechanical deformation test for five ATRON modules connected in a horizontal chain using four connectors. Top: FEM analysis displacement plot. Displacement is exaggerated for visualization. The color shows the displacement of each element caused by gravitational pull. Deformation of the outmost module is about 1.4 mm. Bottom: Real-world deformation test. The horizontal line was put onto the picture after it was taken. Measurements show a displacement of the outmost module of about 3 mm”, from Østergaard *et al.* [33]

Mobile Architectures

Mobile architectures (also *Fluid* architectures) are modular robots which allows self-division into smaller independent modular robots that can coordinate to achieve some task using *swarm* algorithms. When searching for an object, exercising this capability and reduce time spent searching, however there is no guarantee the separated modular robots can connect back together again if one gets stuck or loses the ability to communicate. The *mobile* classification is not mutually exclusive to the above architecture types.

Homogeneous Compositions

Homogeneous composition refers to a modular robot constructed from only a single type/design of a module. This makes for simpler reconfiguration planning as a planner does not have to account for which different modules are located where.

Heterogeneous Compositions

Heterogeneous composition refers to a modular robot of two or more modules types. Planning is more difficult as mentioned above, although the combination of different modules with differently placed degrees of freedom and connector interfaces can give rise to 'exotic' behavior that could not be replicated by a modular robot made from either module design alone.

2.3 Hardware Review

A review of existing module designs is in order at this point, to compare certain differences in module design and the ramifications those have in modular robot abilities verified by experimental results. As mentioned in the problem statement for this thesis (sec. 1.2) it is often difficult to predict how a module's design will affect the multi-module robot's abilities, such as efficiency of reconfiguration. Though by examination of how existing designs correspond to experimented multi-module robot performances, a picture can slowly be built that gives at least some indication how certain module design choices affect the macroscopic performances of modular robots.

This section focuses mostly on connector types and degrees of freedom in a module, as these features determine a robot's reconfiguration abilities. A good survey of 17 module designs was compiled by Østergaard *et al.* that compared some of the geometrical, electrical and physical properties of modules in order to formulate prudent design choices for their own module ATRON [33]. However since this 2006 survey additional module designs such as Superbot (2006) and Roombot (2009) have been developed. This section concludes with a closer look at four selected designs; ATRON, MTRAN (3^{rd} ed.), Superbot and Roombot. The ATRON (2006) and MTRAN (2000) designs represent some of the more well known archetype design concepts that have been produced, extensively tested and cited. Superbot (2006) and Roombots (2009) hold special significance for this thesis. Superbot is the first module type to be tested with this thesis planner, chosen for its many degrees of freedom (3), and Roombot's are the first module hardware we hope to test this planner with, outlined in chapter 6.

2.3.1 What Constitutes a Good Design?

Several factors constitute a good module design. Like many things, mechanical simplicity is often favorable to reduce cost and increase reliability.

Structural Integrity

Structural integrity is an important design factor in any type of robot to cope with any straining forces and torques, and modular robots are no exception. Good designs incorporate rigid modules with stiff actuators to hold connected modules in place. The difficulty that arises in lattice-type modules is that modules need to remain with certain lattice-positioning tolerances in order to avoid collisions and be properly aligned to form connections. An example of a cantilevered beam on ATRONS modules was seen in Fig. 2.1 where the end module from a chain of five had diverged 3mm from its nominated position. Mechanical strain in instances like this is a result of both the rigidity of the module's shell and the stiffness of a connection made between modules. Connection strength is commonly the limiting factor or a robot's overall stiffness and maximum tensile strength [28], and thus is a critical part of a module's design. Structural integrity minimizes module deviations from lattice frameworks but equally important is the design considerations that permits larger tolerances of deviation.

Reconfiguration Efficiency

Reconfiguration efficiency of a robot is dependent on module geometry, connector placement and degree-of-freedom (DOF) placement. Module geometry encompasses the lattice type it may conform to if any. The designers of MTRAN support the theory that double-cube bipartite module designs provide more efficient reconfigurations than single-cube monopartite modules [28]. Connector placement determines how a planner must ensure global connectivity at every step of a reconfiguration. Symmetric connector placement is often preferred by planners, as this can be used to reduce the search space. Usually the more connectors the better as a rule of thumb (sec. 2.3.6). DOF placement determines the morphology or a module; its 'kinematic options'. Modules that only have DOFs parallel to their connectors are unable to change their own orientation and require neighboring modules to help them do so [28, 33] (the issue of *flavor*, discussed sec. 2.3.3). As another rule of thumb; usually the more degrees of freedom the better, as shown by Superbot's performance over MTRAN owing to its additional central DOF that MTRAN does not have.

2.3.2 Connection Mechanisms

Reliable connection mechanisms have been one of the most challenging aspects of modular robotics to get right. The most dominate methods in the short history of modular robots since CEBOT have to use magnets (permanent and electric) and/or pin & hole assemblies. The use of magnets, especially electromagnets which can be controlled without any moving parts is always an aesthetic design proposition however its main drawback is the continual consumption of power to maintain an attachment. Pin & hole connection arrangements only ever consume power when actively disengaging or engaging new modules. Their construction however is mechanically complex, increasingly so if modular robotic research heads in the direction of miniaturization towards micro and nano scales. Correct alignment is an important concern for type of connector, but magnetic connectors can be rather forgiving by absorbing small positional and angular misalignment errors between connecting surfaces [28, p.438].

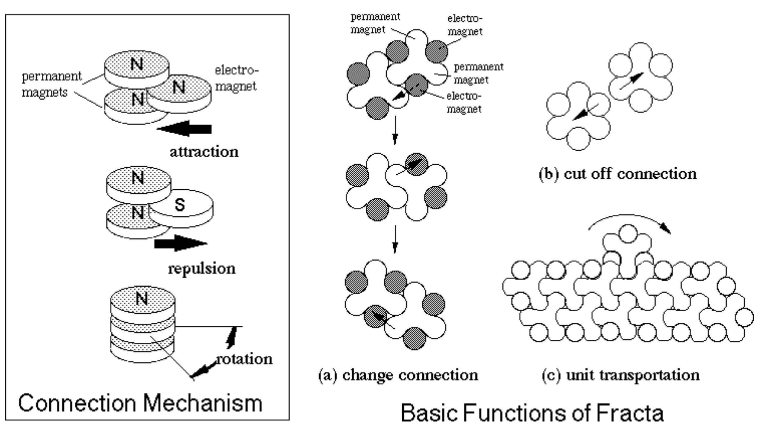
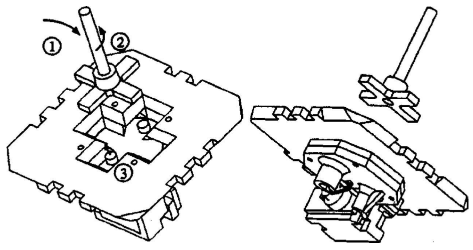
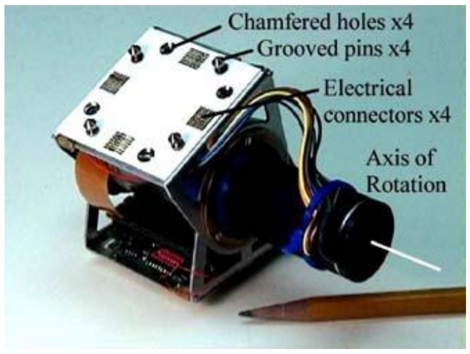


Figure 2.2: Left: Fracta's connection mechanism, Right: Same mechanism used for locomotion, from Østergaard *et al.* [33]

The Fracta module design [26] is unique in that its electro/permanent magnetic connection mechanisms are also used for module locomotion, repelling or attracting itself to neighboring modules as seen in Fig. 2.2. As such it has no need for any DOF, and therefore no actuators to control the DOFs, resulting in a very simple (albeit power hungry) design. Purely mechanical assemblies of locks/hooks/grooves have been more popularly however, with modules *Metamorphic* [9], *Crystalline* [38], *Micro-Unit* [54], *Chobie* [21], *CONRO* [8], *Polybot* [51], *M-TRAN* (3rd gen.) [28], *3D-Unit* [27], *Molecule* [24], *I-Cube* [46], *ATRON* [33] compared to five module designs listed in the Østergaard's paper [33] using magnetic connectors. Out of these 11 mechanical connector designs, only *Polybot* and *3D-Unit* include hermaphroditic connectors. At first glance hermaphroditic connectors may seem more attractive for planning purposes, albeit more complex mechanically, as planning programs can always safely assume (i.e. no need to verify) that any two connectors are able to form a connection. Happily, lattice regularity can prevent accidental same-gender alignment from ever occurring by implicitly branding certain lattice cells as male only and adjacent cells female only. The designers of MTRAN published findings of this phenomenon [29], and is discussed in sec. 2.3.4.



(a) I-Cube Connector



(b) Polybot G2 Connector

Figure 2.3: Two mechanical connector designs. Left: I-Cube's connectors that inserts and rotates a pronged pin to lock itself to a female connector. Right: Polybot G2's hermaphroditic connectors, using 4 pin/hole pairs over a greater surface area provides increased torsional stiffness and redundancy, from Østergaard *et al.* [33]

2.3.3 Case 1: ATRON

ATRON is a monopartite lattice-based module of two hemispheres separated by an actuated axle. This axle is its only degree of freedom, so to reorient it relies on neighboring modules to have perpendicular axle directions, which can spin them to face a new direction. An ATRON robot must have modules facing in all 3 Cartesian coordinate directions (x, y, z) to be fully reconfigurable because of this effect. The designers referred to the 3 possible orientations of modules as the *flavors* of modules, a robot would be made of x-ATRON, y-ATRON and z-ATRON flavored modules.

The axle between both hemispheres is designed for infinite revolution. At the 'equator' of the module lies 5 concentric slip rings to allow for continuous transmission of power and data between hemispheres (both hemispheres contain microchips) as they spin relative to each other. The ATRON shape is ellipsoidal but by virtue of its connectors placements it packs like a sphere. The designers opted for a less well known spherical packing pattern they called "Rhenium Oxide" named after the pattern ReO_3 forms naturally in crystals. It

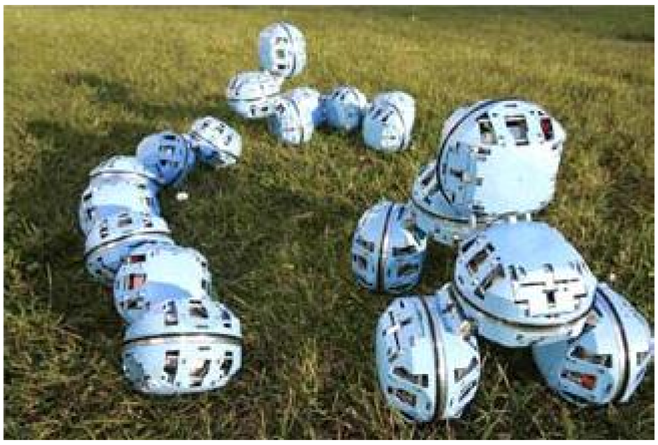


Figure 2.4: ATRON robots shown in snake and buggy morphologies, from Østergaard *et al.* [33]

has a packing efficiency of 55.54% and has 8 equidistant connection points to other spheres (modules) around it.

Designers of any connections interfaces generally choose one of two geometric types of connector; multiple connection points spread over a *surface-to-surface* alignment of two modules, or a single *point-to-point* connection as spheres do. *surface-to-surface* provides greater torsional stiffness, though *point-to-point* allows more movement options in close proximity of other modules. For example spheres are free to spin when packed tightly together however cubes cannot. The ATRON design is an attempt to compromise between both connection interface types, for the freedom of movement advantage and at least some torsional stiffness. This is impossible to do if module-module connections first require flush alignment of modules because *surface-to-surface* interfaces immediately collide with one another. The ATRON design gets around this problem by separating connected modules by a few millimeters as shown in Fig. 2.1. This separation is achieved by retractible latches forming external connection that is offset from the module surface. Each of the 8 connection interfaces has a 3pin/hole connection points as shown in Fig. 2.5, the minimum

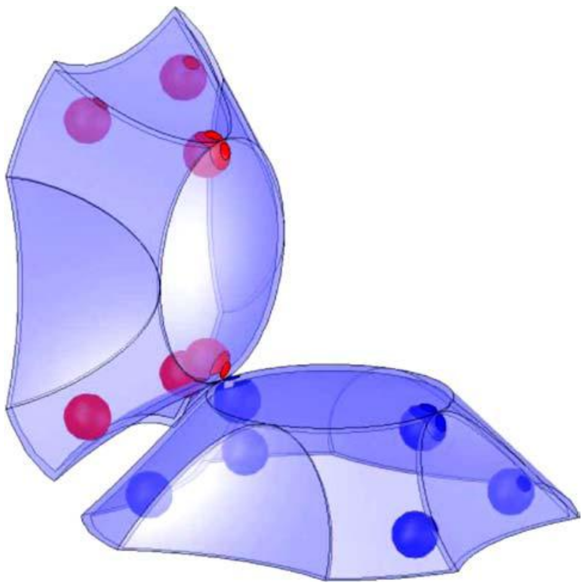


Figure 2.5: ATRON Connector Positions, from Østergaard *et al.* [33]

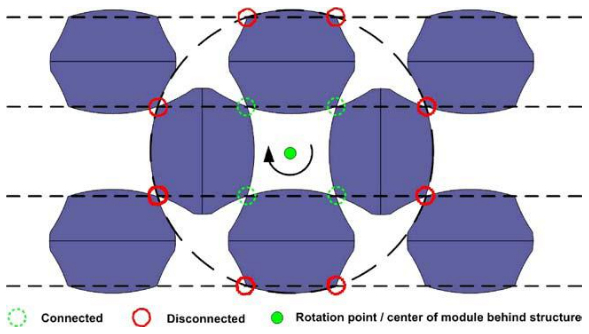


Figure 2.6: ATRON Internal Looping Movement Possible, from Østergaard *et al.* [33]

number of connection points to withstand torques in any direction.

One interesting property of the ATRON design is that collective motion of modules is permitted within a fully packed structure of ATRONs. Fig. 2.6 shows how this might happen; the four modules within the dashed circle can be turned 90 degrees by the rotation of a 5th module they are mutually connected that has its axis direction facing into the page.

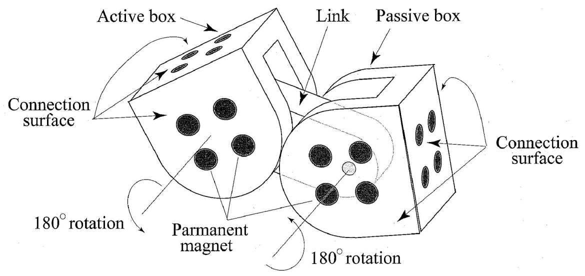


Figure 2.7: MTRAN Schematic, from Murata *et al.* [28]

2.3.4 Case 2: MTRAN

MTRAN [28, 29] is a bipartite hybrid module, both parts are semi-cylindrical and each fills one cubic cell within the robot's cubic-lattice framework. Both parts are connected by a common 'link'. Fig. 2.7 shows a schematic of this. Dimensionally MTRAN is 66×66×132mm, runs on a 12V power supply, has a mass of 400g and use PI microcontrollers that communicate via asynchronous serial 4800bps connections. Only one module in a cluster is connected to a power supply, and shares power to the other modules through the connection interfaces. Thus a MTRAN module cluster must always stay connected and is a *non-mobile* architecture module [53, p.912].

MTRAN has 2 axis of rotation (degrees of freedom), one located in each part. The semi-cylindrical shape of each part allows them a 180 degree range of rotation. Both parts are physically the same except for connector genders, and so both axles in either part are parallel. This contributes to the same reconfiguration consideration discussed in ATRON which required different *flavors* of modules (different module-axis orientations) to reorient modules. Otherwise, reorientation is impossible and a MTRAN-based SRR cannot reconfigure.

This module, developed at the MEL research institute in Japan has three generations;

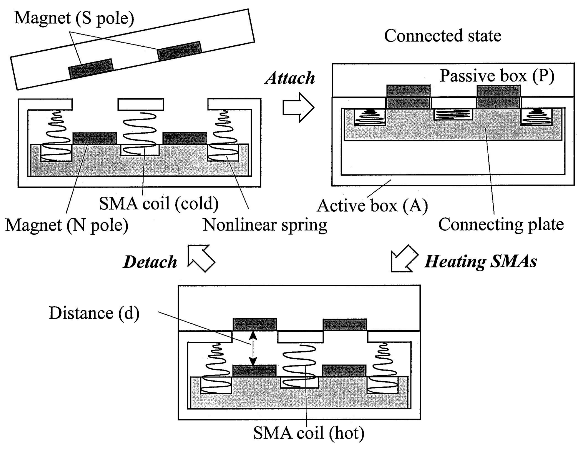


Figure 2.8: MTRAN I and II connection mechanism, from Murata *et al.* [28]

MTRAN I and II used a used a magnetic connector type, the third generation opted for a pin & hole assembly. The magnetic connectors, as seen in Fig. 2.8, are made up of a combination of permanent magnets, springs and shape memory alloy (SMA) coils. Connections are formed by mutual attraction of permanent magnets from 2 modules, and is resisted and dampened in part by non-linear springs that are engineering to be slightly weaker than the magnetic force (about 25N). This force difference can be controlled by the SMA coils which are heated to increase net repulsive force of the modules, and when it has made up the magnet-spring force difference; the modules will detach. This however can take 5-15s to heat the coils and an additional 20-30s cool down period before able to form new connections.

A magnetic connection between 2 connectors involves an *active* and *passive* connector. The active is so named because it both controls the connection using the SMA coils, and thus the only connector that consumes power. All MTRAN designs have 3 connectors located over each part. The magnetic-connector generations of MTRAN (I and II) have one ‘active’ part, of which all 3 connectors are active. The other part is completely passive.

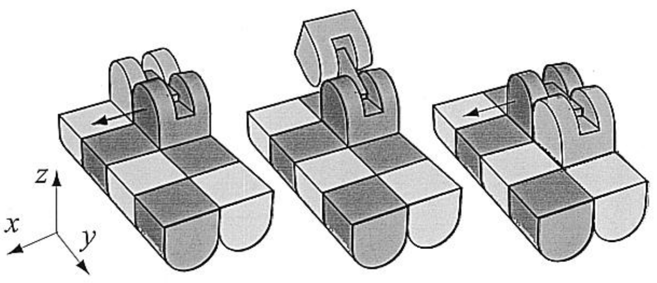
The pin & hole MTRAN III mimics this asymmetry with one part having 3 male connectors and the other part purely female connectors.

Same-Gender Connection Avoidance

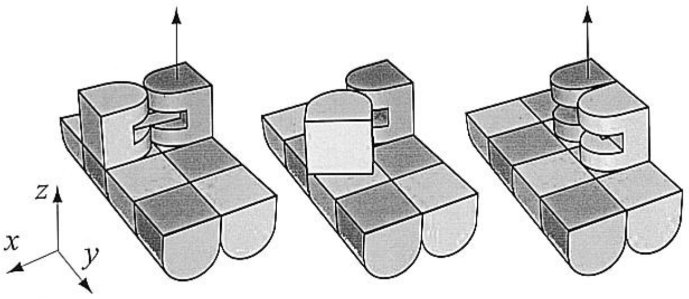
Intuition may lead one to believe that for planning purposes (and reconfigurations in general); it is a disadvantage to have gendered connections instead of non-gendered/hermaphroditic module connection interfaces. This is because any one connector is henceforth restricted to forming connections with only half the available connectors around it. As the designers of MTRAN found out, by virtue of a bipartite module within a lattice framework, this never turns out to be a problem, because no sequence of moves could ever align two same-gendered connectors anyway, even if they tried [29].

This is perhaps more clearly seen in Fig. 2.9, showing the three basic modes of locomotion a MTRAN module can exercise. Two modes are self-moves seen in Fig. 2.9a & Fig. 2.9b, where the module can roll forwards or pivot sideways independently, and Fig. 2.9c shows an example of cooperative behavior (sec. 4.4.1), where one module helps relocate another. The middle pictures of both Fig. 2.9a & Fig. 2.9b show a module in mid movement, but every other configuration depiction shows the modules conforming to the lattice structure. Notice that white module is always located above a black module, no matter how it moves, once in a new lattice cell, there are only black modules which is can connect to, and using these set of moves it is actually impossible to align itself with a same-gendered connector, given that all black connector genders will be opposite to all white connector genders. Thus just as a checker piece can never change the color of cells it is adjacent to, neither can a module part of a bipartite lattice-module.

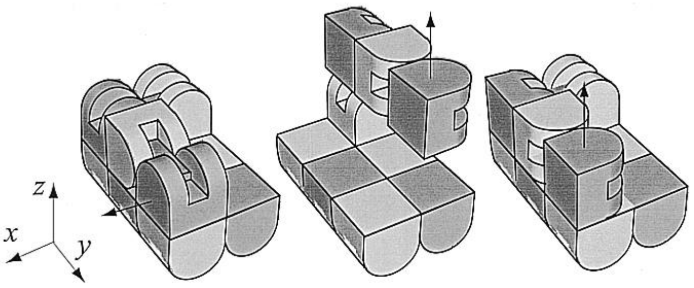
The designers of ATRON were able to design their module in full knowledge of the



(a) Forward Roll, axis in x direction



(b) Pivot Translation, axis in z-direction



(c) Mode Conversion, axis in y-direction

Figure 2.9: MTRAN native methods of locomotion, from Murata *et al.* [28]

previous MTRAN design, however theirs is a *monopartite* module. To the author's knowledge, there are two ways lattice regularity can still be exploited for monopartite module designs:

1. By using the same mentality of the bipartite module case, which effectively brands certain cells as 'female only' and adjacent cells 'male only' much like a 3D checker board. For monopartite modules, they could be manufactured in two populations of modules, one of male connectors only, and one purely female. This is however less appealing as it requires the presence of members of both populations in the makeup of a robot

2. ATRON's method of *gender parity* where each module half has and two female two male, where same genders are placed opposite (180 degrees) from each other. In addition, modules of different flavors (axle directions) need to adopt a cyclic gender-connector positioning policy, such as placing female connectors at the:
   - x-direction sides of y-ATRONs
   - y-direction sides of z-ATRONs
   - z-direction sides of x-ATRONs

   The *gender parity* is defined as 'zero' if conforming to the above, and 'one' if not. A 90 degree rotation will change gender parity of a module half, including any other module halves connected to it that revolved with it. So an even amount of module-half movements is required to conserve gender parity, and by doing so an ATRON robot can reconfigure without connection gender mismatches

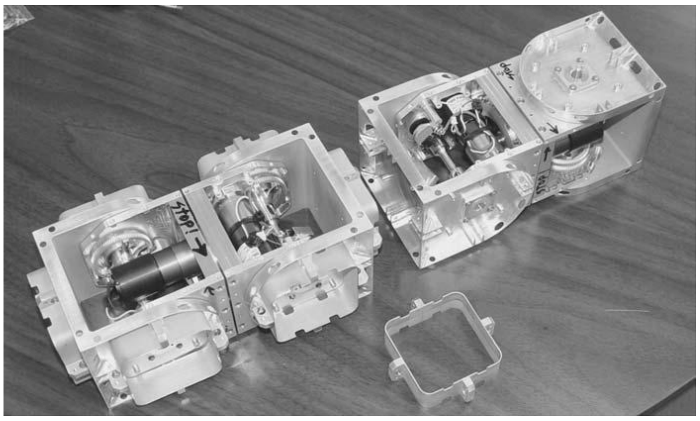


Figure 2.10: Two Superbot Modules. Leftmost module is oriented like a MTRAN module, the rightmost is like a CONRO module, from Shen *et al.* [40]

2.3.5 Case 3: Superbot

Superbot [40] is a very similar design to MTRAN and CONRO from which the designers drew much inspiration. The major difference is an additional central axle connecting both module parts which is infinitely revolvable, providing a third degree of freedom. This central axle allows the Superbot to act either a MTRAN (part axis aligned) or a CONRO (part axis perpendicular) module as seen in Fig. 2.10. Because Superbot has at least 2 degrees of freedom that are not mutually parallel, the *flavor* concept requiring differently orientated modules does not apply. Thus it is possible for Superbot-based SRR which begins in a state where all modules are facing the same direction to break the conformity and reorient modules to face differing directions.

Connector placement (Fig. 2.11) is like MTRAN. The designers stated their mechanical connection mechanism are genderless and also allow "considerable" tolerance for module angle misalignment when forming connections, though details have not been published [40, p.169]. The designers have also conducted some impressive experiments of gait movements, showing how a Superbot robot can move like a snake, caterpillar, spider and even a

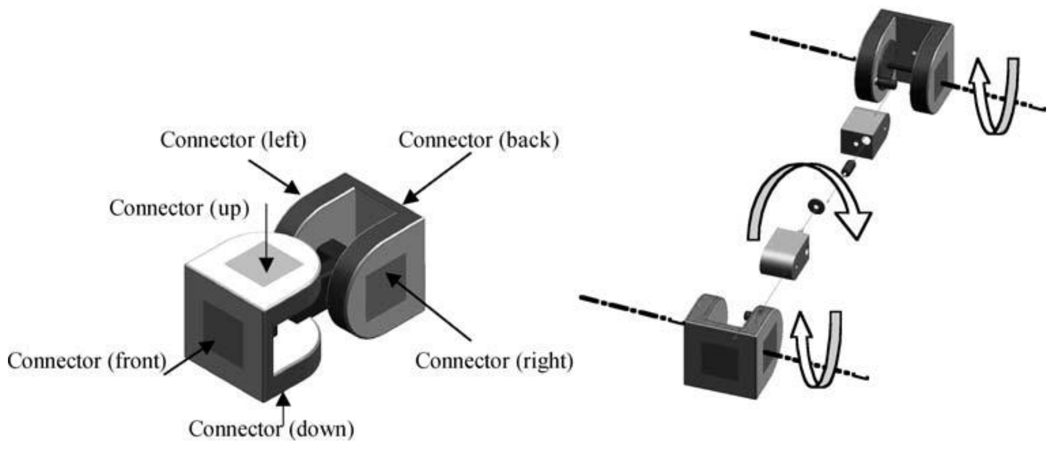


Figure 2.11: Design of Superbot. Left: connector placements. Right: Exploded view showing 3 joints that are the degrees of freedom, from Shen *et al.* [40]

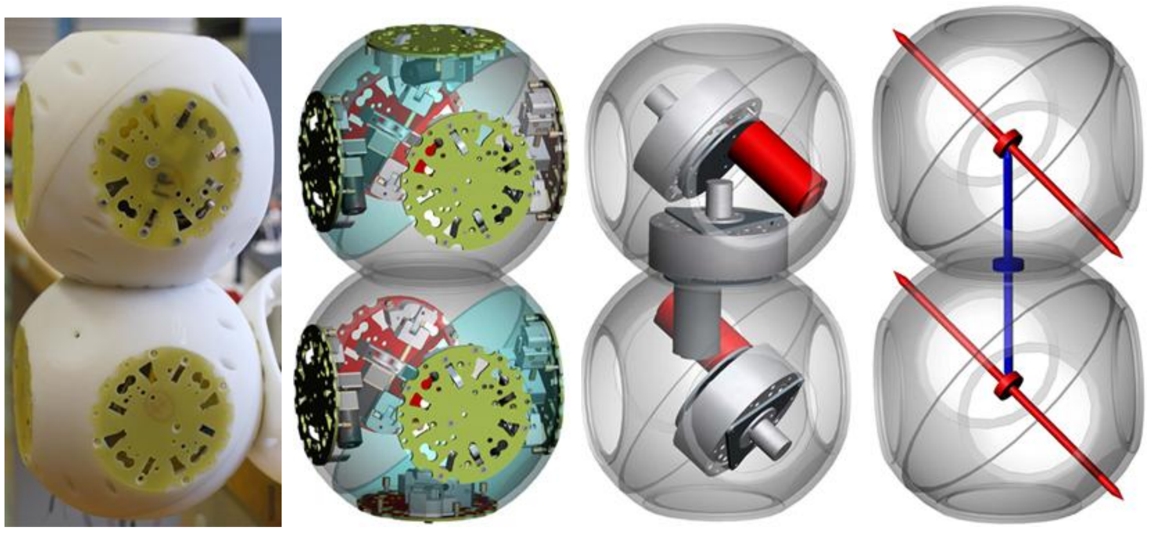
rolling track that can propel itself at an average speed of 1.0m/s for over 500m on battery power [22].

2.3.6 Case 4: Roombot

Roombots [41], like Superbot, are hybrid modules that conform to a cubic-lattice structure. They compose of two cubic parts with highly rounded edges as seen in Fig. 2.12a.

The name ‘Roombot’ was developed as part of the designers’ vision for these modules to be used within residential settings as transformable furniture. They propose that such furniture could be static or interactive, such as a chair that walks or climbs stairs whilst someone is seated on it. Presumably the module’s curvy ABS plastic surface and non-pinching morphology (Fig. 2.12) would make them more suitable to regular human contact than modules like MTRAN or Superbot.

A great benefit of Roombots is they have the option to be fitted with 10 hermaphroditic active connection interfaces, so a connector is present on every face of the double-cube



(a) One Roombots module (b) Up to 10 ACMs (c) 3 main Motors/gearboxes (d) 3 axes of rotation

Figure 2.12: Roombot Schematics, from Sproewitz *et al.* [41]

shaped module. Not only does this enable increased flexibility of reconfiguration and packing options for a planner, is also reduces planning complexity (space and time) by:

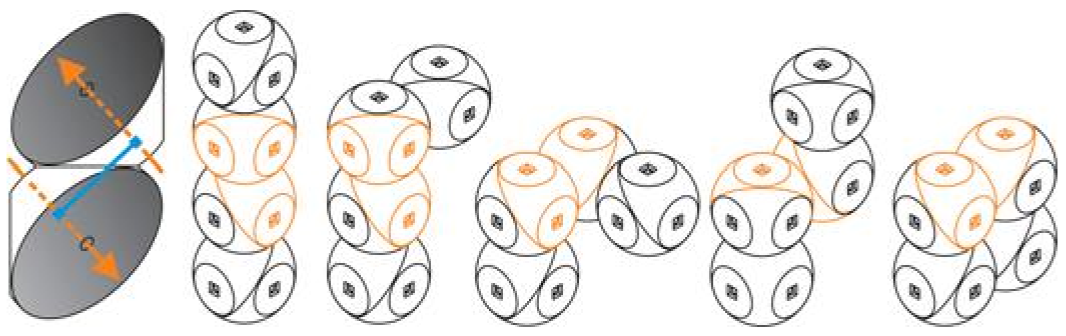
- Increasing the number of symmetries in the module, which means less considerations required from a planner, and therefore reducing the searchable state space size (sec. 4.2.2)
- Surface moving modules can safely assume any side of the robot's exterior is a possible connection point, it does not need to enquire where another module's connectors are specifically located
- When considering a desired shape to reconfigure into, a planner can assume such a robot will be completely connected if every module is adjacent to at least one other module, which is a very fast & simple check. Otherwise if connectors exist on only some of the module faces (such as Superbot which has 6 connectable faces out of 10), a conclusive check on a goal shape's global connectivity is not so straight

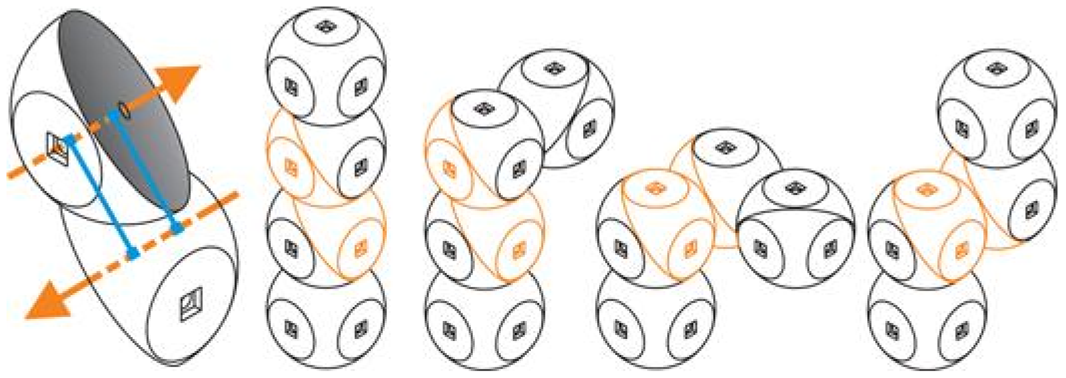
forward. A whole host of robot configurations will satisfy the given goal shape, some will conserve global connectivity and some will not due to different connector placements (Roombot does not have this problem, every configuration it can assume within a given goal shape results in the same connector placements, namely every face of its cubic parts). A planner may have to iteratively check many configurations before it finds one that satisfies global connectivity unless a special purpose routine can be developed that can somehow returns one globally connected configuration if one exists for a given shape

Roombots have 3 degrees of freedom (Fig. 2.12d), each making use of slip rings to be infinitely revolvable. Superbot in comparison has only one degree of freedom that is infinitely revolvable (its central axle) whilst its other two axles are restricted to rotations within a 180 degree range.

Their connection interfaces are hermaphroditic by virtue of a four-way symmetry of pins & holes (Fig. 2.12a). Currently the designers are deciding between the benefits of using 2 lathes or 4 per connection interface. Latches are built with fiber reinforced plastics, which allows the 2 latch option to carry up to 16kg. This helps meet a primary design goal which is that one module should always be able to move 2 additional connected modules whenever it revolves about any of its degrees of freedom. This gives Roombots the ability to form chain-arms of 3 modules long, with 9 axes of rotation total, enabling it can act as a short manipulator that can position an end effector at any point in space within its reach and be oriented in any desired direction. The minimum torque constraint this imposes on module actuators, given each module is $1.4kg$ and $220mm \times 110mm \times 110m$ is $1.16Nm$. To overcome this, all actuator DC motor-gearbox combinations installed are rated to $5.0Nm$ of torque.

Perhaps the most distinctive of Roombot's feature is its arrangement of degrees of freedom as seen in Fig. 2.12d. Modules like MTRAN or Superbot in contrast are made using axles that are either parallel or perpendicular to each other which an intuitive arrangement for modules that operate within a cubic lattice framework. Roombot itself still conforms to a cubic lattice however both its part-axles are 45 degrees offset from its central axle connecting both parts. This gives rise to some interesting and unique morphologies as seen in Fig. 2.13. These three figures depict different morphologies possible with two Roombot modules placed end-to-end in three different ways. Morphologies like these can be discovered by native kinematic planners which take into account the inherent design of a module, and thus are able to take full advantage of morphological options a module design offers. Native kinematic program implementation is therefore usually more complex by considering these specifics, but the pay-off is its potential to discover far more efficient reconfiguration strategies than a more generic planner. Generic (non-native) planners are less specialized but can be made compatible across several modules designs by restricting module morphology options to 'standard actions'; common actions that every module type the planner uses is able to perform. An example of this is extending MTRAN's planner for the control of a Superbot based robot. A Superbot module has all the functional options a MTRAN module has, yet also has a central axle (which MTRAN does not have). The MTRAN planner can be used for both MTRAN and Superbot by simply not actuating Superbot's central axle, effectively making it a MTRAN module. In this case, the actions a MTRAN module can perform are deemed the 'standard actions' for both modules. So even though this planner performs legal reconfigurations for a Superbot SRR, it forsakes the use of additional degrees of freedom, thus morphological options available to it, which could otherwise help reconfigure the robot in less atomic actions saving time and power.


(a) Axes orientation skew


(b) Axes orientation parallel

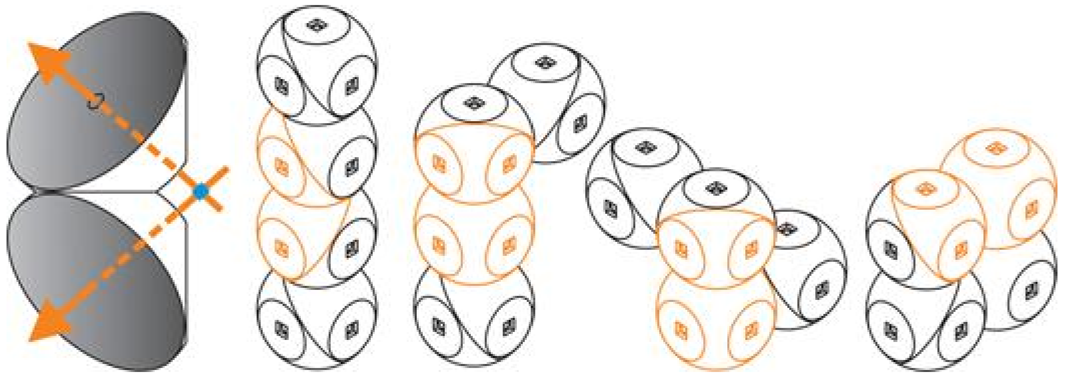

(c) Axes orientation orthogonal

Figure 2.13: "Possible grid-reconfigurations with two Roombots modules connected in-series. The resulting shapes depend on the axis-orientation of the two center blocks, colored in orange: (a) Skew: 5 options, I-, L-, 3DS-, S- and U-shape. (b) Parallel, 4 options. (c) Orthogonal, 4 options", from Sproewitz *et al.* [41]

Chapter 3

Background: Reconfiguration Planning

This chapter presents some of the underlying theory of reconfiguration planning to give the reader some background information on this field before introducing a new design in the following two chapters. This chapter begins with an introduction of common SRP concepts, and progresses onto *reinforcement learning* which is receiving increasing attention in the SRR field as a valid means of reconfiguration planning. A survey & comparison of several existing planners is included for the purpose of identifying effective techniques previously tested. This includes a discussion of the implications of the various SRP design considerations surveyed. This follows with more detailed case studies of the five SRPs surveyed. The review of these five particular planners does not constitute as a comprehensive review of planners in general, but their mutually differing methodologies and originality of designs illustrate the current variety of existing SRPs. These case SRPs are:

1. Fracta Planner (1994)
2. MTRAN Planner (2002)
3. Claytronics Planner (2006)

4. Million Module March (2008)
5. Graph Signature (2008)

The Fracta, MTRAN & Claytonic planners are all purpose-built planners (unique to one module design), developed by the respective designers of each module. None of these developers gave explicit names to their planning algorithms, so they are referred to by their module name in this paper. Fracta pioneered a lot of the reconfiguration planning field with a strictly localized communication planner that relies on a simple yet novel self-assembling reconfiguration process that is highly distributed. MTRAN is a well known module and used a combination of hierarchical search and meta-module stacking to search effectively and limit its own search space. The Claytronics planner opted for an unintuitive approach of using *holes* like semiconductor physics (the absence of modules) for robot reconfigurations. The last two cases are algorithms developed independently of the module designers. Million Module march presents an efficient way of coordinating many generalized module motions in parallel by the use of a special navigation function from reinforcement learning, and Graph Search looks at using aspects from graph theory to represent and solve reconfigurations of a MTRAN-based modular robot.

3.1 Introduction

A modular robot's ability to change shape is derived from a rearrangement process of the multiple interconnecting modules which can attach and detach from each other. Reconfiguration planning is concerned with how to order such module movements such that an SRR can reconfigure from shape-A into shape-B. This section discusses some of the common aspects and challenges inherent to reconfiguration planning.

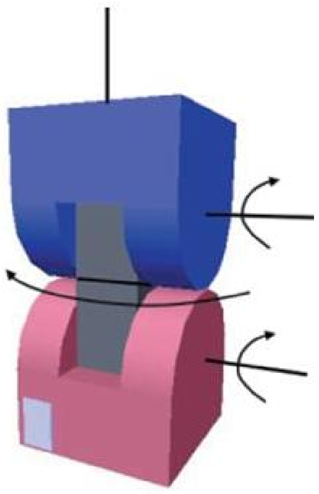


Figure 3.1: A module 3-R module with three degrees of freedom, from Fitch and Butler [14]

3.1.1 Native Kinematics

A reconfiguration planner needs to be able to plan module movements for the type of module the SRR system is based on. To do this the planner needs to be able to glean two important aspects of the design of the module it is considering:

1. Native Kinematics: from a particular configuration, where can a module possibly move to next?
2. State Representation: what information is needed to fully define a module's geometry and kinematic options in 3D space?

Both these aspects are dependent on the module's geometry and where its degrees of freedom lie. To plan for a module that will reconnect and detach old connections as it moves along the surface of the robotics structure then knowing the placement of the connector interfaces is also crucial. The number of degrees of freedom can range from zero such as in MEL's *Fracta* module to six in another of MEL's module the *3D-Unit* [33, p.168].

Fig. 3.1 is an example 3-R module made up of two parts, each the same, connected by a central axle, twisting which is one of the degrees of freedom. The remaining two degrees of freedom are pins in either part placed perpendicular to the central pin. These allow both parts to alter the direction they are facing. When both the native kinematics and the state representation are defined, a planner is able to search through a module's native kinematic space for solutions as to how a module can move to another location, or reorient itself as desired.

3.1.2 Decentralized Nature

Ultimately the algorithm needs to run onboard an SRR itself which acts as a multiprocessor machine being composed of many modules each with a respective microchip(s). To take full advantage of this, programs concerned with the robot's whole need to be decentralized in order to run in parallel over the many microchips. The most ideal case, is a planning algorithm that can be decentralized to the point that each module need only consider its local space when determining its next movement, and be independent of the global configuration of the SRR so as to minimize long distance module-module communication which incur significant time delays. In this case there would be no limit to the amount to modules allowed in the system, as the computational complexity of running the program grows linearly with the amount to microchips available to execute it.

3.1.3 Connectivity Concept

Connectivity is the physical linkage of modules. Global connectivity refers to a modular robot that is structurally a single piece by virtue of the various individual connections linking composing modules. Global connectivity is a binary statement; a modular robot

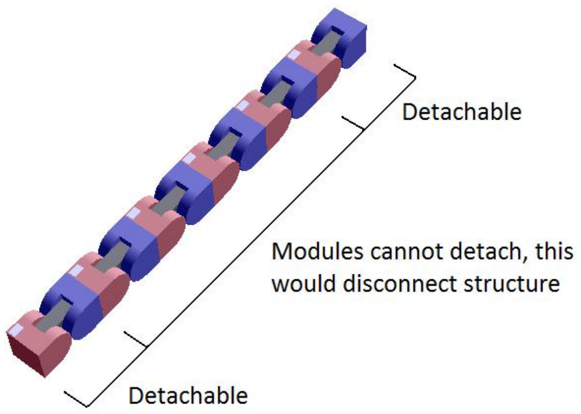
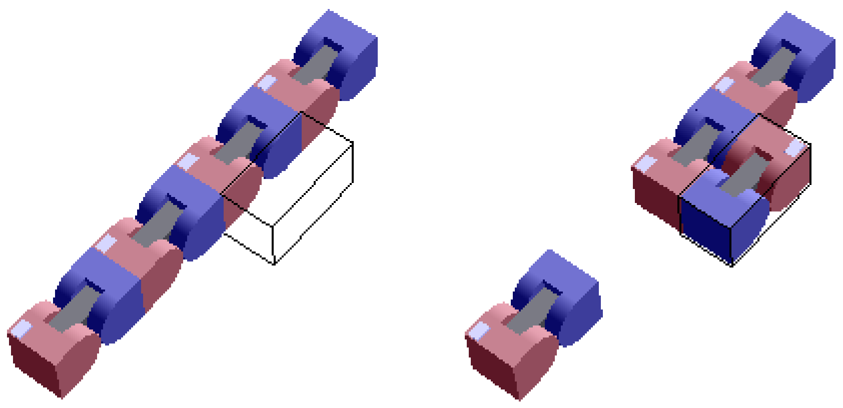


Figure 3.2: Connectivity Concept

is either globally connected (one piece), or it is not (meaning it is in multiple separated clusters).

A connectivity verification function is often required by a planner to determine if a certain module is able to uproot and travel to a different location in the robotics structure without disconnection of the global structure. Global disconnection is usually extremely undesirable as it is not guaranteed the robot can joint back together again, especially if the robot is lattice defined and the result of a disconnection leaves two robot segments in separate planes separated by several degrees. In this case the most connections mechanism will be unable to reconnect if module faces are not already parallel and may not be able to recover from the situation. This situation is especially undesirable if modules rely on connectivity as a sole means for power distribution or (hard-wired) communication.

Figure 3.2 shows which modules are able to uproot and become mobile from a straight line structure of 6 Superbot modules. Here if either end-module is able to be detached and independently move over the structure however if any of the four middle modules did the same thing global disconnection would be violated. Figure 3.3 shows an example of this



(a) (b) Global connectivity violated

Figure 3.3: (a) a modular robot attempts to fill wireframe box with one of its own modules, (b) it cannot use a middle module as this would violate global connectivity

disconnection. If the configuration in Fig. 3.3a attempted to fill the wireframe box with one of the modules and selected a module in the middle to do so, this would result in the two globally disconnected structures shown in Fig. 3.3b. This is a potentially unrecoverable situation.

3.1.4 Common Challenges

Good Heuristic Functions

Most Self-Reconfiguration Planners (SRP) incorporate heuristics; metrics that quantify the difference between two configurations of a robot. SRP's use heuristics as a feedback mechanism to guide search algorithms from a robot's current configuration to the final by roughly indicating if a particular search path is searching down set of states that are increasingly

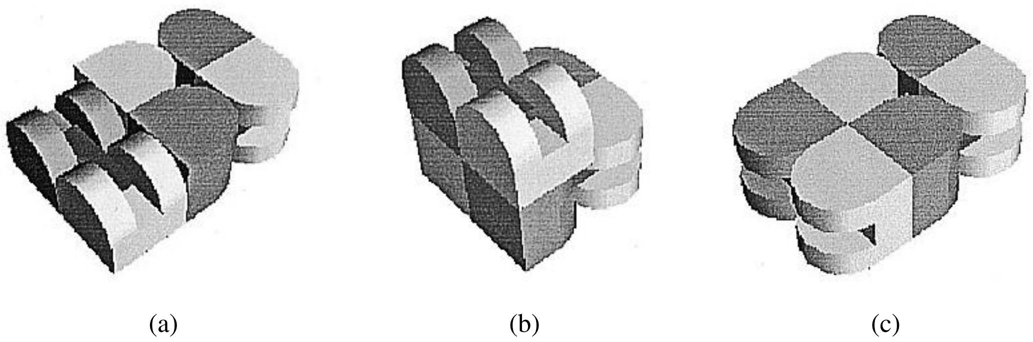


Figure 3.4: Reconfiguring MTRAN: Resting on flat ground in which this robot can move over, it is possible to reconfigure from configuration (a) to (b) but not (a) to (c), from Murata *et al.* [28]

distant from the goal configuration or not. A good[1] heuristic is key to an efficient algorithm, however as the MTRAN designers found out is often very hard to develop one for 3D anisotropic module designs such as the MTRAN module. They found that "for most 2-D lattice systems and for isotropic 3-D modules, there is a good correspondence between distance in lattice space and distance evaluated as the number of necessary motion steps. Therefore, the lattice distance can be used as a metric for those systems and it gives planning methods at reasonable cost" [28]. However they found that this relationship definitely did not flow onto the 3D anisotropic module case and in fact "simple lattice distance gives almost no information in the M-TRAN system" [28], MTRAN being a 3D anisotropic module.

This can be seen in figure 3.4 which shows some possible reconfigurations of three MTRAN units. A MTRAN robot cannot always reconfigure between different configurations exampled by Fig. 3.4a and Fig. 3.4c due to the problem of flavors (sec. 2.3.3), but for

[1]a 'good' heuristic is defined as one which most reliably informs the planner as to the true reconfigurable distance (number of atomic actions required) to transform one configuration into another. A perfect heuristic will inform this distance exactly, a poor heuristic will report a distance that often very different from the true distance but can still serve as somewhat of an indication

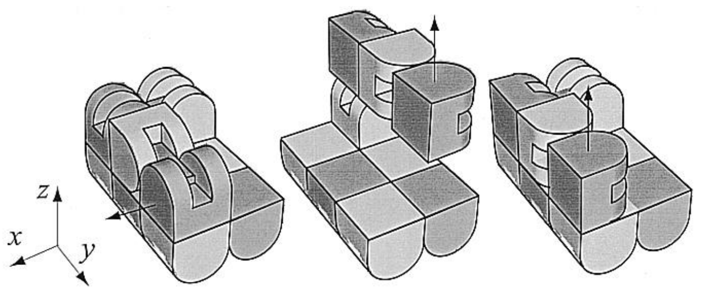


Figure 3.5: MTRAN pivot translation (rotation about Y direction). A single atomic action results in 6 adjacent cell movements, from Murata *et al.* [28]

the reconfigurations which are possible (Figs. 3.4a to 3.4b) it takes 15 atomic actions for the lower-left module to move up on top of the structure. Both parts have only moved 2 adjacent cell spaces each, so 4 adjacent cell movements has translated to 15 atomic actions in this instance. In contrast, figure 3.5 shows as just 1 atomic action is executed, it results in 6 adjacent cell movements. Thus a direct relationship between lattice distance (adjacent cell movements) and atomic actions required to reconfigure is almost non-existent and makes for an extremely poor heuristic.

Instead more inventive and often anti-intuitive heuristic definitions are required. Later this chapter explores some case planners and how they have developed heuristics, often tailored to a particular module design. These range from analyzing patterns of a module's connecting surfaces as to which are connected at any one time (Fracta), to borrowing concepts found in graph theory (Graph Signature), and comparisons of current & desired surface topologies of the modular cluster (Claytronics).

Collision Avoidance

Collision detection is often difficult to design for and can be expensive to compute but nevertheless mandatory for an SRP [1, p.869] [28, p.440]. Collision detection routines require geometric models of a robot and a complete understanding of a module's kinematics to predict collision events. The level of resolution these models are designed with determines the reliably of the prediction. If the SRR is lattice based then collision detection can be computed more effectively; the 3D world is naturally segmented into discrete regions which either do or do not contain a module and a module's kinematics can be modeled by a set relative lattice-cells the module encroaches for particular actions performed. Mechanical strains can result in modules moving outside their lattice defined positions (sec. 2.2.1) which complicates the detection process and remains an open problem.

3.1.5 Hierarchical Methods

Reconfiguration planning is a search task based in configuration space. The addition of every module not only increases this space exponentially, but also increases its dimensionality owing to its degrees of freedom. The end result is a vast, highly dimensional space. Hierarchical methods are a means of breaking down a search task into more manageable subtasks to improve the chances of finding a solution, albeit not necessarily the optimal solution. Hierarchical method have been proposed as a means of solving reconfigurations [5, 14, 28, 34], especially for deterministic planners. A natural structuring of a hierarchical search involves one subroutine that determines appropriate relocations of modules, and relocates them by routinely calling on a lower level subroutine to discover valid actions sets that successfully relocate the module to its given goal position.

Reinforcement learning also poses several benefits to reconfiguration planning, whereby a robot can steadily improve on the quality of reconfiguration solutions it computes. Hierarchical reinforcement learning has been studied and offers accelerated learning opportunities in large spaces. This hold potential application for reconfiguration learning [4, 12], discussed in section 3.2.4.

3.2 Reinforcement Learning

Reinforcement Learning (RL) has been proposed as a means to assist robot reconfigurations whereby modules gradually learn the complete morphology of one module or a collection of modules by exploring all action-options available [42, 47]. Such learning methods can reduce planning complexity and search space problems by relying on modules to inevitably discover action protocols (the 'details') that lead to correct global reconfigurations. A good survey of reinforcement learning and its applications can be found in Sutton and Barto's book (1998) [44]. This thesis is not wholly concerned with reinforcement learning but does use value iteration methods from dynamic programming, a subbranch of RL, to construct a navigation function that modules use for optimal path planning.

3.2.1 Overview

Reinforcement learning is a sub-branch of machine learning concerned with learning via unsupervised interaction with an environment. Supervised learning is a large field itself, where a *learning agent* can be trained to make better decisions with the help of an external & more knowledgeable supervisor such as a human technician or higher level program. This type of learning begins with a set of simulated training examples of input objects and

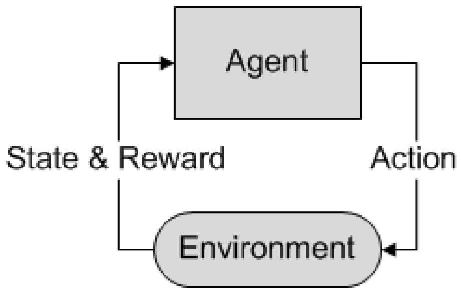


Figure 3.6: Interaction between an agent and its environment, from Sutton and Barto [44]

desired outputs, as the agent attempts each example it can be given feedback straight away as to its performance so as to calibrate itself for next time. Examples include classification algorithms, artificial neural networks and pattern recognition. Unsupervised learning however does not have this luxury. In this case the learning agent must learn from its mistakes directly. Unsupervised learning is the process of interacting with an environment and observing the consequences.

Reinforcement learning deals more specifically with goal-seeking agents, and without supervision, learn how to optimize their behaviors to reach their goal. Different actions performed in different circumstances will often lead to different results (a queen piece moving forward one square in a chess game may or may not be a good move; it depends entirely on the current state of the game). Common terminology in this field refers to these ‘circumstances’ as the *state* of an agent. The state is an agent is the sum total of all its knowledge about itself that has any relation to it achieving its goal. I.e. for a robot in the field with a goal location to travel to, its state representation may need to include its current position, what obstacles are in its immediate vicinity, predicted weather conditions etc. Its ability to observe that it has been painted a grey color however may not be relevant to it achieving this goal, in which case would not be included in the state representation.

In reinforcement learning, an agent must explore different actions available to it under many different states to establish a rule-set or *policy* of how to behave in the future in order to reach its goal as efficiently as possible. This policy is the mapping of each state to an action it should execute when in that state. Sutton and Barto describe this as akin to stimulus-response rules in biological systems [44, p.7]. A good policy is one that will consistently pick an action that results in the agent being in a more desirable state than where it was before, where a ‘desirable state’ is defined by being closer to its goal. A goal is something that is inputted by a human or higher level program which tells the agent what it is looking for or a value it should be working out how to optimize. What constitutes a goal is widely varied. It can be a location in 3D space, a vehicle’s fuel consumption or ‘checkmate’ in chess.

In reinforcement learning a reward function must exist that calculates the *desirability* of any state encountered, a single number which reflects a state’s desirability. In the example of a field robot above, any state inside the physical goal location would be given a ‘good’ reward of say ‘+1’, any state not inside the goal location could be ‘0’, and for state from which the robot would be completely stuck ‘$-\infty$’ would be a reasonable choice. The agent can use the reward function to compare different states and its policy can become: ‘pick the action that will lead to the state of highest reward’. This will definitely help the agent when it is one atomic action away from being in the goal location, or falling into a ditch, because it will be presented with a choice of different desirability values of which it can greedily choose. However the reward function cannot be used in every situation, because when far from a goal or any hazards than all actions will lead to states of ‘0’ desirability (because it will still be far from a goal or hazard). All actions being equal, it cannot make an informed decision which action to take.

Ultimately the agent should not be selecting actions that optimize its immediate reward, but those which maximize its expected total accumulation of rewards over time, i.e. it should be thinking long-term. For the field robot, this would mean getting to the goal location as quickly as possible, as only when there can it start accumulating rewards. This concept of ‘expected accumulation of reward over time’ is what is called a *value function* [44, p.8]. Once the agent has explored the environment enough that it knows at least one path from the current state to the goal location, a value function will be useful in guiding the agent back to the goal location. The value of each state along this path to the goal location will be greater than the previous one, as this state is one action-transition closer to being able to start accumulating rewards. In this way an agent’s policy can simply be a greedy choice of which action leads to the states with highest value.

To express the value function mathematically, first the return function, R_t, needs to be expressed. The *return* is defined as the sum of rewards, r_i, after time t up until time T where time is segmented in discrete intervals called *episodes*;

$$R_t = \sum_{k=1}^{T} r_{t+k}$$

To incorporate the idea that rewards are better if received sooner rather than later (prompting an agent operate in reasonable time), a discount factor γ is included which devalues the present valuation of a future rewards (depending now how many episodes in the future it lies). The discounted reward is;

$$R_t = \sum_{k=1}^{T} \gamma^k r_{t+k}$$

.

The value function of a particular state is the *expectation* of the discounted reward function under a given policy;

$$V^{\pi}(s) = E_{\pi}[R_t | s_t = s]$$

Where E is the expectation function, π is the decision-making *policy* and s_t is the state of the agent at time t. Some manipulation of the above formulae results in the Bellman equations defining the value function V;

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')]$$

Where $\pi(s,a)$ is the agent's policy; the probability of taking action a when in state s, $P^{a}_{ss'}$ is the probability of arriving at state s' given the agent is at state s and chooses action a, and $R^{a}_{ss'}$ is the expected reward given the agent is at state s chooses action a and transitions the state s'. The policy which achieves the most reward is referred to as the *optimal policy*. The expected reward of the optimal policy from any state is greater or equal to that of any other policy. This is the *optimal value function*;

$$V^{*}(s) = \max_{a} \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{*}(s')]$$

3.2.2 Exploration vs. Exploitation

A major trade-off that exists in reinforcement learning is between *exploration* and *exploitation*. Exploring an environment allows the agent to build up a model of its surroundings. A model is a knowledge of possible states the agent can be in and something that simulates the environment to predict, given a state and action, what new state the agent would likely end up in. This gives an agent the ability to plan paths to goals without having to physically try them all again. More exploration gives the agent a larger model, and a more complete picture of options available to it.

Say a field robot is collecting and depositing a resource between two locations and every time it deposits a unit resource it receives a '+1' reward. After an initial purely exploration period, it finds one path through the environment linking both locations, and can now start transferring resources and collecting its rewards. At this point the agent can either continue to 'exploit' its current knowledge; collecting rewards along the fastest route it knows of, or it can explore more terrain in the hope of finding a shorter path which will increase future productivity. Conversely, the extra exploration could turn out to be a complete waste of time. Arguably the best techniques discovered is to balance this tradeoff; concentrating purely on a exploration based policy during an initial period, and slowly an exploitation policy takes more precedence as time goes by. This is because the agent becomes more and more likely to have found the optimal path between two states. Eventually exploitation becomes the dominant concern of the agent, and time allocated to exploration tapers off to a minimum threshold, perhaps 10% (by not reducing this value to 0%, the agent retains the ability to readapt if the environment changes for whatever reason).

3.2.3 Dynamic Programming

Dynamic programming is one of the fundamental classes of reinforcement learning algorithms which compute optimal policies given a perfect model[2] of the environment (such as a chess game where a player's current state and environment is completely known). This is

[2]the model of the environment is a tool the agent uses which predicts how the environment will respond to certain actions. A perfect model always predicts the environment's responses exactly

in contrast to Monte Carlo Methods or Temporal-Difference Learning classes which begin learning with incomplete or no such models of their environment.

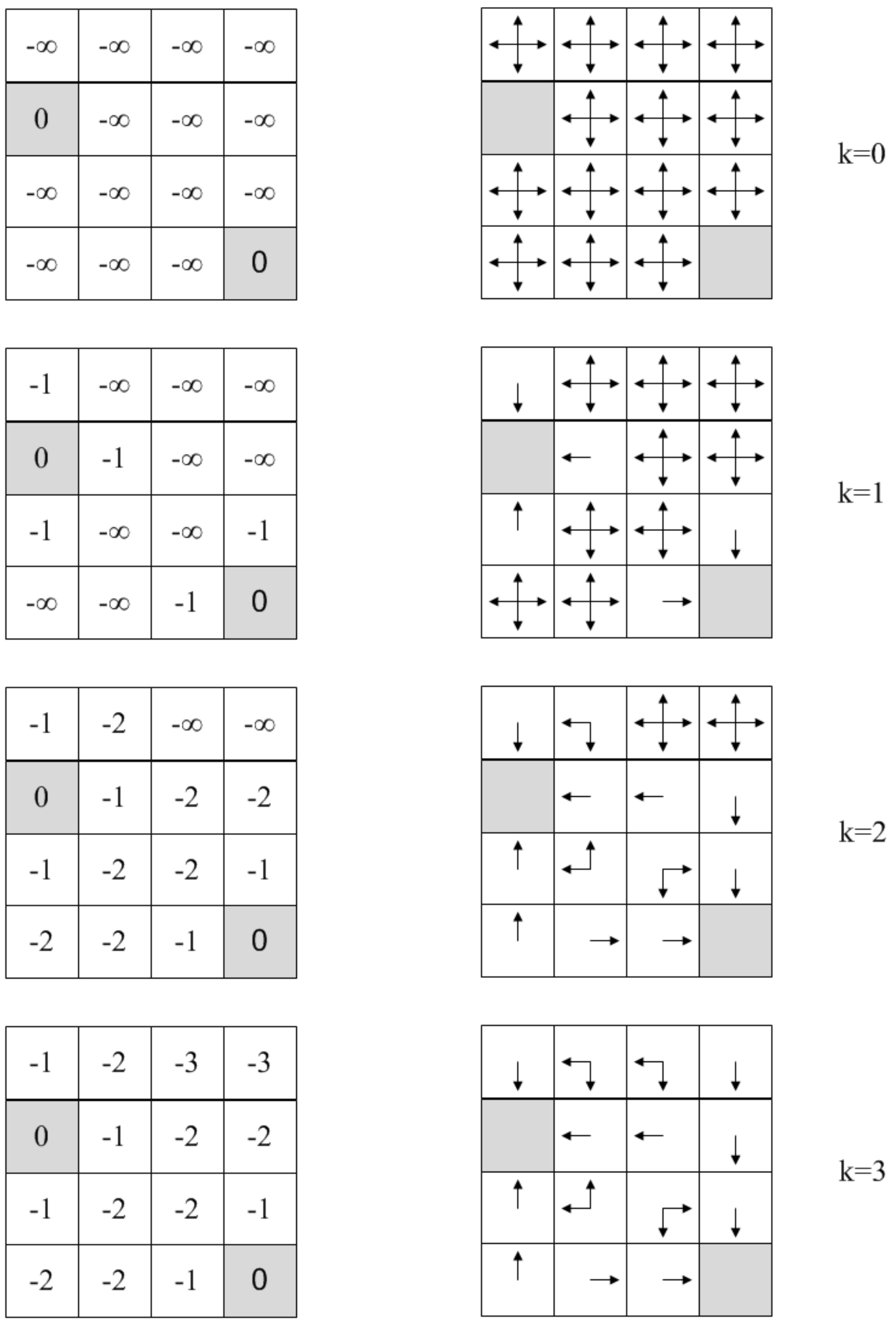
Algorithm 3.1 Dynamic programming: Iterative policy evaluation, (edited) from Sutton and Barto [44]

```
1: Initialize $V(s) = -\infty$, for all $s \epsilon S$
2: **repeat**
3:     $\Delta \leftarrow 0$
4:     **for** each $s \epsilon S$ **do**
5:         $v \leftarrow V(s)$
6:         $V(s) \leftarrow max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
7:         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
8:     **end for**
9: **until** $\Delta < \theta$ (a small positive number)
10: Output: $\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
```

Optimum policy evaluation is main goal of dynamic programming. Optimum policies are not probabilistic[3], they make well defined decisions. The optimum policy function function $\pi(s)$ is the action a the agent performs when in state s to perform optimally. Algorithm 3.1 shows the iterative method of computing a value function called *policy evaluation*. Policy evaluation begins initiating the set of all states (S) with a default value of value $-\infty$ except for goal states which retains fixed values of 0. Once all state values have stabilized (condition on line 9), the optimum policy is outputted which is a greedy search of the highest value state (line 10). At every iteration (repeat/until loop; lines 2 to 9) every state value is updated according to the previous value of states in its neighborhood (line 6). It selects the maximum value-reward number from neighboring states[4].

[3]except in the cases of multiple equally good options, it may make a random choice of these

[4]This update is weighted by the probability of each possible state s' the action a could lead to from state s with the term $\sum_{s'} P^a_{ss'}$. Some cases are purely deterministic (such as chess), and which case this terms would disappear (it equals one)



(a) Policy evaluation (b) Policy improvement

Figure 3.7: Optimum policy evaluation & improvement; an example where grey cells indicate goal regions (value 0) and an agent in any other cell can move horizontally or vertically one cell per episode

An example of an evolving value function is shown in Fig. 3.7a and corresponding policy improvement in Fig. 3.7b for successive iterations (denoted by k). In this example an agent can be in any cell, but can only move horizontally or vertically one cell per episode. The grey cells are goal regions. The policy evaluation produces a value function (Fig. 3.7a) which the agent can use to navigate towards a goal by using a greedy policy of moving to cells of increasing value. However each move incurs a reward of '-1' (the $R^a_{ss'}$ term of algorithm 3.1 line 6 is would be a constant in this example) to represent the cost of energy and time expended doing so. Beginning at the fist iteration (k=0), all values are negative infinity except for the goals. By the next iteration each state (cell) updates by the maximum value of '$reward + neighborValue$' it finds. The cells directly adjacent to the goal cells immediately become $-1 + 0 = -1$ whilst cells further away remain at negative infinity because all their neighbors are still negative infinity. As the next iteration follows, the states adjacent to '-1' cells become '-2' and so on. Finally the navigation function stabilized at the third iteration and the corresponding optimum greedy policy in Fig. 3.7b is outputted. By following this policy and agent anywhere on the grid will find its way to one of the goal state in an optimum amount of transitions (multiple arrows indicate equally good choices).

3.2.4 MAXQ

A problem that arises in non-hierarchical dynamic programming is the propagations of value functions as seen in Fig. 3.7a can be very computationally expensive for large state spaces. The value functions of future rewards can take a long time to propagate outwards to where an agent may be currently located. Hierarchical algorithms on the other hand have the ability of abstracting knowledge from certain actions of certain states in the state space and applying that knowledge to other states without necessarily having to re-learn action-implications at the new location. This is akin to function approximation technique where the model of an environment may not be perfect, but it requires much less computation than a perfect model. MAXQ [12] by Dietterich (1999) is arguably the state of the art of hierarchical dynamic planners.

As an example, reusing the grid example of Fig. 3.7, an agent can be in any of these 16 locations however this time its orientation matters as well; it can either faces up, down, left or right. There are thus $16 \times 4 = 64$ possible location-orientation states. An agent can either choose to spin clockwise (changing its orientation) or move one cell forwards in the direction it currently faces. A flat propagation technique could be used as the original example did, and 64 states would have to be assessed individually. A hierarchical method can exploits state abstractions by decomposing actions available to all states into separate subroutines to share information between states via macros it develops. An example is that seen in figure 3.8. This begins by using the movement decision from the previous example (up down, left or right). Because it has to face a direction in this example before it can move in that direction it calls on the 'Orient Correctly' subroutine which spins the agents until in the desired orientation. Notice this subroutine can be applied to any location in the grid, so when MAXQ learns this phenomenon from one state it can apply it to other states and *share* knowledge across the state space. In this way the maximum number of states the algorithm needs to encounter are $16 + 4$ not 16×4 in order to operate 'intelligently' in the gird, and thus learning has been accelerated. A separation of actions also allows the agent to ignore large amounts of the state space information and focus on what is relevant when in any particular subroutine.

The downside of this technique is it cannot always guarantee global optimality (like

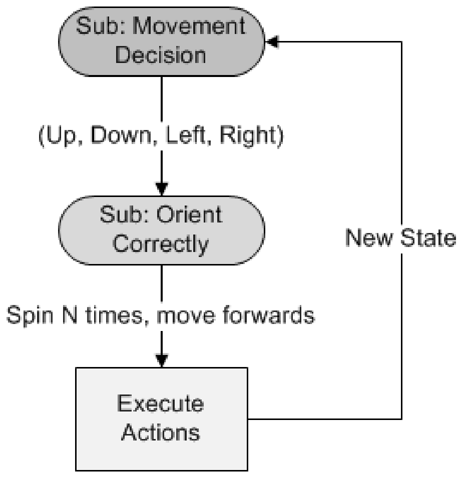


Figure 3.8: Subroutine hierarchy example of MAXQ

flat propagation can), even though it can guarantee that each subroutine execute with local optimality. Each subroutine level has an associated value function which is uses to make greedy optimal decisions from. In this example, the ‘Movement Decision’ subroutine has a value function exactly that seen in the previous example, Fig. 3.9a. The ‘Orient Correctly’ subroutines value function is just the minimum of the (negative) number of clockwise 90 degree turn required until the agent faces in the desired direction. If the agent is located and oriented as symbolized by the arrow in Fig. 3.9b, the ‘Movement Decision’ subroutine will opt to move 2 cells upwards (locally optimal number of movements). However when it executes this, to move upwards the first time the ‘Orient Correctly’ subroutine must spin 3 times clockwise before it points upwards (also locally optimum). So all in all there are 5 actions taken total. So it did find a solution to getting to a goal location, however it did not find the globally optimum solution which is to move left 3 times without spinning (only 3 actions total). ‘Movement Decision’ did not decide to do this because the goal upwards was only 2 cells away as opposed to 3.

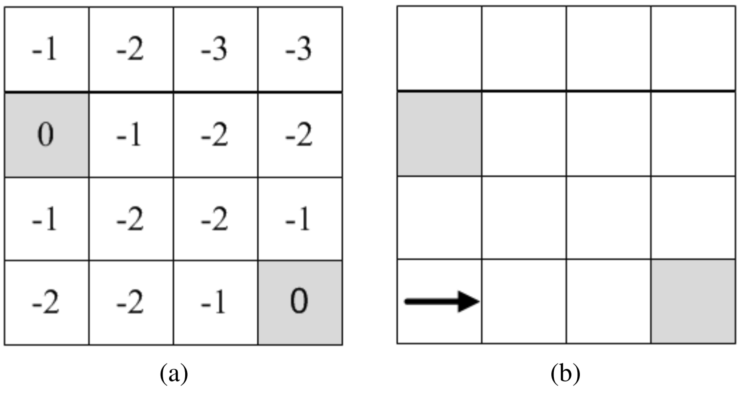


Figure 3.9: Example of MAXQ without global optimality

The benefits MAXQ offers is the accelerated learning of state space navigation. Such benefits could potentially be applied to reconfiguration problems of modular robots due to their large configurable space. If such an algorithm could be implemented in this field it could potentially make a much faster reconfiguration planner that would additionally be able to autonomously *learn* better reconfiguration strategies as time goes by. The only downside is global optimality of reconfigurations are not guaranteed. However global optimality is usually not a realistic design goal for any SRP and so a close-to-optimal solution would still be a very welcome result.

3.3 Comparison of Existing Planners

This section highlights some major algorithmic decisions that exist for SRP designs. Table 3.1 compares five existing SRPs. This follows with discussions of implications and trade-offs inherent to each principal classification of SRP design. All five SRPs listed are further detailed as case studies after this section, completing this chapter.

Table 3.1: Comparison of Selected Reconfiguration Planners

Planner	**Fracta**	**MTRAN**	**Claytronics**	**MMM**	**Graph Signature**
Cite	[26]	[28]	[36]	[14]	[1]
Year	1994	2002	2006	2008	2008
Module Type	Fracta	MTRAN	Claytronic	gen.	MTRAN
Lattice	hexagonal (2D)	cubic (3D)	hexagonal (2D)	cubic (3D)	cubic (3D)
Max. modules helper aids	0	1	0[3]	0	∞
Process	stochastic	deterministic	stochastic	stochastic	stochastic
Module Composition	homogeneous	homogeneous	homogeneous	homogeneous	heterogeneous[4]
Computing Architecture	decentralized	centralized	decentralized	decentralized	centralized
Module Movements	parallel	serial	parallel	parallel	serial[1]
Metamodules?	modules	modules[5]	metamodules[2]	metamodules	modules
Simultaneous Actuations?	n/a	simultaneous	n/a	n/a	not simultaneous
Time Complexity / module			linear	sub-linear	exponential
Space Complexity / module			linear	linear	

[1]Not explicitly mentioned in their paper, but they stated (p.6) their configuration space grew by roughly a factor of $16 = 2^{4modules}$ from every configuration, this is characteristic of a serial planner

[2]The *holes* are deemed metamodules, as they are a ‘structured arrangement of (absent) modules’, just as ordinary metamodules are structured arrangements of modules

[3]a 2005 Claytronics publication [19] anticipated it could carry neighbors, the 2006 publication [36] and subsequent publications did not mention it. Assumed the idea was abandoned

[4]Discussed planning with module of both gendered connections, and non-gender connections. Connectable interfaces are an important aspect of a module the planner needs to consider, therefore this is considered heterogeneous composition

[5]This planner uses metamodules for robot static composition, not for locomotion however

3.3.1 Stochastic vs. Deterministic

A simple definition of a stochastic program is; 'a program which makes a decision based on a randomly generated value at least once in its execution'. A deterministic program will always produce the same output if its inputted values remain unchanged. Modeling a planning program's performance on random variables is not as useless as it may first seem, in fact Asadpour *et al.* report most approaches to SRPs use stochastic-based optimization methods such as Simulated Annealing or Genetic Algorithms [1, p.864]. Stochastic programs can be effective in this context because the search space can be so large and the heuristics functions can be so uninformative, that the search environment of an SRP bears great resemblance to an environment based on probability distribution functions. Stochastic programs excel in these probabilistic environments because they are willing to *explore* (sec. 3.2.1) their environment by occasionally searching in directions that would otherwise seem like bad decisions, due to the random element. However because the environment is (seemingly) probabilistic, the what initially seemed like a 'bad decision' to the planner can sometimes turn out to be a really good one, of which the planner takes note and updates its probabilistic model of the world it is in. This is something a deterministic planner typically never does, bent on making 'good decisions' continually.

The attraction of deterministic planners lies in their reliability; they can find a reconfigurable solution between two configurations, they guarantee they will always be able to do so again if required. However to *guarantee* this, the SRP often needs to know everything about a robot, all of the time, such that nothing unpredictable happens when it executes (everything is foreseen). This requires much coding to consider everything logically, and when installed on an SRR, usually requires the SRR to share large volumes of information between modules, to constantly be aware of its global state. The disadvantage of deterministic planners, is they are characteristically more demanding of computation resources and therefore often slower to execute. Thus higher *complexity* is the tradeoff cost for *reliability* when designing a planner.

The advantage of stochastic planners is that they are often very simple and easy to implement. Modules can be programmed to take actions based on just their own state and and that of any immediate neighbors (Fracta Planner; sec. 3.4), or perhaps that of neighbor's neighbors as well maximum (Claytronics Planner; sec. 3.6). This kind of reconfiguration is really a result of self-assemblies, each module acting independently, but in such a way that the global cluster converges onto a shape goal unbeknownst to any module. Because a stochastic algorithm is usually of a simpler implementation than a deterministic one, it most likely lacks full knowledge of the entire robotic cluster, and therefore a module will often not know the long-term consequences of taking a certain action. So the searchable state space is no longer perceived as deterministic (which it is) by the stochastic planner, instead each state-action pair[5] are considered to have an associated probability density function as to the desirability of their final outcome. So, the state space only 'seems' to be probabilistic to a stochastic planner due to its own naivety, a module will not consider it present state as much as a deterministic planner would have it do in order to predict the exact result of its actions. The advantage of programming a stochastic planner, is that it is not necessary to confront the entire complexity of the search space head on, stochastic methods have some ability to 'learn'[6] about which actions are usually better to take, and to some degree the module's can figure out their own way to the goal shape. So programming is simpler as the programmer does not have to comprehensively consider many types of situations a

[5]meaning; performing a certain action from a certain state

[6]by building up a probabilistic model of its environment in order to make better decisions in the future

robot can be in. The disadvantage of this, is that is that stochastic planner's can potentially blindly lead their robot into configurable stalemates (Fracta; sec. 3.4, Claytronics; sec. 3.6) due to poor global planning. It is always possible to identify and avoid these undesirable situations with planner which makes more globally based decisions, i.e. robot-defined decisions, of which deterministic planners are usually closer to than the other (localized) extreme; purely module-based decision making.

3.3.2 Metamodules vs. Modules

Metamodules are a structured arrangement of multiple modules. They can be considered a single module themselves that are a different *type* of module than the modules they are composed of. An example of a 4-module MTRAN metamodule is shown in Fig. 3.13a. There are several reasons why a planner may opt to collate modules at its disposal into a smaller number of metamodules to then plan for. Two main reasons include:

1. **Generalization**: If a planner is designed to reconfigure modules of a common shape, such as a cube, then it has the potential to be applied across many different module designs, if those module designs can form metamodules of that common shape. Such a planner that is almost[7] hard-ware independent is advantageous in that it is portable; it requires little or no re-design when executed on different robots of varying module designs. Million Module March (sec. 3.7) is one of the planners studied below that plans for generalized metamodules.

[7]it still depends on modules being able to form a metamodule of the common shape, and each such metamodule needs to able to perform collective motion primitives the planner assumes of a metamodule, such as sideways translations. Depending on the module design and metamodule arrangement, a metamodule may not always be able to perform both these tasks, in which case it cannot be planned for by this SRP

2. **Size reduction of the configuration space**: Metamodules reduce the planner's configuration space in two ways. Firstly there are less individual elements to consider. In the case of the 4-module metamodules above, the number of individual elements is quartered. Secondly, metamodules will (most likely) have less configurations possible that a single module. The 4-module MTRAN metamodule for example does not have any degrees of freedom to orient itself about. The only way its configurable description can be changed is by relocating it, or reorienting it. In fact the MTRAN planner (sec. 3.5) further restricts metamodule orientation, so the planner can always safely assume that the metamodule is in its one standardized configuration, and only needs to consider its position when planning for them. Comparatively a Superbot module has 1728 unique configurations it can be in for a given position (sec. 4.3). Both these factors contribute to a significant exponential reduction of the searchable space for a planner, allowing more rapid execution

A great disadvantage of using metamodules over modules is the losing the ability to explore and exploit every feature of a module's design in computing an optimal or close-to-optimal reconfiguration solution. Even though a 'metamodule-planner' can potentially compute reconfigurations solutions faster than a 'module-planner', it will most likely be much more inefficient a solution. Additionally the time saved by a metamodule-planner finding its solution quickly may not contribute to time saved overall, as module movements are comparatively slow, usually the order of seconds. So the more time invested by a module-planner will probably translate to a faster physical robot reconfiguration and save time overall. Another disadvantage of using metamodules is the loss of resolution of the shapes a robot can assume. If a robot's metamodule composes a high number of modules, the robot can become very 'pixilated', reducing its ability to perform certain tasks.

3.3.3 Homogeneous vs. Heterogeneous

Homogeneous and heterogeneous module compositions have already been discussed in sections 2.2.1 & 2.2.1. The five SRPs discussed in this chapter only plan for homogeneous composition except Graph Signature (sec. 3.8). Heterogeneous robots require extra consideration for a planner, meaning added programming effort and enlarged searchable space, because it must account for which different modules are located where. A homogeneous planner never needs to distinguish between modules, it knows (without verification) they are all exactly the same. As an example of the effect heterogeneity has on the size of an SRP's searchable space; Fig. 4.11b shows how 32 bipartite modules can be packed together. Taking into account only position and orientation of modules (not the max. configuration options possible) there are 5.05 billion unique ways of forming this cube shape, a small fraction of this robot's overall configuration space. However, if the composition changed to using bipartite modules of two types, 16 of module A and 16 of module B say, this number would increase by ${}^{32}C_{16}$ to become 3.03×10^{18}, a 601 million fold increase[8]. So heterogeneous planning does requires more computational complexity, but it can improve SRR application, by carrying additional modules that contain specialized sensors, or solar panels etc.

3.3.4 Centralized vs. Decentralized

Decentralized SRPs are arguably better in many respects than centralized SRPs, and more desirable for modular robots. Centralized computation is the execution of a program on a single microprocessor, decentralized computation is the execution of a program over multiple microprocessors. Decentralization always incurs several programming complexities

[8]The use of 'C' in '${}^{32}C_{16}$' represents the mathematical *choose* function

such as synchronizing programming threads, avoiding race conditions and avoiding deadlock situations where different threads wait on each other. Though ultimately, for SRP to be effective, it must be decentralized as modular robots have many microchips (standard module designs include at least one microchip in each module). Thus a decentralized SRP take full advantage of a modular robot's natural computing architecture, and allows for scalable[9] algorithms. If an algorithms time-complexity is linearly proportional with the amount of modules or less (e.g. Claytronic sec. 3.6), Million module March sec. 3.7), then such a program would never be limited by the number of modules in a robot, because for every new module added, the added complexity that adds to the system is matched by the addition computation power the robot receives.

3.3.5 Serial vs. Parallel

A parallel SRP is able to plan simultaneous motions of modules. Parallelized motions of modules allow much faster robot reconfigurations and are required for robots of many modules to reconfigure in a reasonable amount of time. Parallelized planning is therefore preferred and necessarily for the reconfiguration of SRRs of many modules in reasonable time. Serial planning is always easier to implement, as there is not as many programming overheads such as dynamic collision detection between multiple moving modules. Instead one module would only need to avoid collision with static modules of known locations, a much simpler feat. Parallelized motions can add to the number of choices available to a planner to consider about how a reconfiguration could be done, but if the planner is locally-oriented (opposed to global knowledge-based decision making), as most stochastic planners are[10], then other module movements are often independent and irrelevant to a

[9]scalable; meaning an SRP can plan for SRRs of many modules
[10]e.g. Fracta sec. 3.4 & Claytronic sec. 3.6

module making a localized decision about how it should move, considering only other module movements within its immediate vicinity[11]. In this case the added number of choices would not necessarily translate to decreased performance due to increased complexity.

Parallelized module movements are required for a robot reconfiguration to scale well with increased modules to the system. Serially reconfiguration of a robot executes in quadratic polynomial time. In contrast parallelized reconfigurations have the potential to execute in sub-linear time as exampled by Million Module March (sec. 3.7). Parallelized programming is therefore more desirable in an SRP.

3.3.6 Simultaneous Actuation

Simultaneous actuation is a module's ability to actuate several of its degrees of freedoms simultaneously. This is physically possible with most module designs, however planners usually limit these actuations to one at a time for simplicity of planning (MTRAN is an exception, sec. 3.5). In most cases, Simultaneous actuation generally does not offer a robot more advantage over consecutive actuations except in a small number of specialized situations. It does increase planning complexity by adding more optional actions a planner can perform on a single module, so for these reasons simultaneous actuation is usually not considered for SRPs nor discussed much in the literature. This aspect of planning was considered in this thesis planner (sec. 4.4.2).

3.3.7 Max. modules helper aids

Many planners include the ability of one module being able to 'carry' another, often referred to a *helper* module, but also a *converter* module (MTRAN sec. 3.5). For some

[11]an exception to this rule is routines for checking global connectivity is not violated by a certain motion

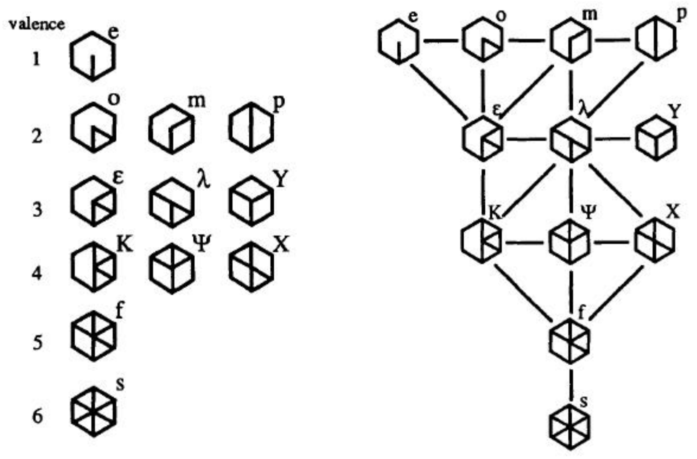
module designs this ability is necessity for the planner to find a possible reconfigurable solution between two configurations of a robot (sec. 4.4.1). This holds true for the MTRAN (sec. 2.3.4), ATRON (sec. 2.3.3) and Superbot (sec. 2.3.5) modules. The 'Max. modules helper aids' refers to how many modules are able to be carried simultaneous by a helper module. This will be defined both by planner design and module torque restrictions.

Helper modules does not necessarily constitute simultaneous actuation, nor parallelized movements between the 'helping' module and the 'helped', they are all wholly independent aspects of an SRP. The centralized implementation of this thesis does include helper modules and simultaneous actuations, and does not support parallelized module motions for example. A more detailed discussion of helpers and their implementation in this thesis is located in sec. 4.4.1.

3.4 Case 1: Fracta Planner (1994)

To the author's knowledge; the Fracta planner [26] represents the first autonomous SRP to appear in the field of modular robots. Based on the Fracta module [26], this planner offers a simple self-assembling reconfiguration algorithm that each module in a cluster executes based entirely on local communications from connected modules. Many other hardware designs incorporate local communication styles and implement package-relaying for communications between modules not directly connected, but the Fracta planner acts even more localized by assuming package-relaying is not possible, and communication really is just limited to connected neighbors.

This planner considers the state of a module in terms of its *connection type*. The Fracta planner is unique in this case, due to not having any degrees of freedom, and so besides



(a) Connection types

(b) Connector-type transition Diagram

Figure 3.10: Fracta connection-type diagrams, from Murata *et al.* [26]

position and orientation[12], connections are the only way of describing a module's state. A Fracta module can connect up to 6 neighbors and has 12 unique ways of being connected (excluding those created by rotation or mirror image) as seen in Fig. 3.10a, connection represented by internal lines. They are depicted as hexagons, as can connect in 6 different places and pack into two-dimensional hexagonal patterns as seen in a simulation goal configuration (bottom shape) in Fig. 3.12.

3.4.1 Representation of a Fracta

Fig. 3.10b shows a 'connection-types' network or transition diagram. Each node represents one of the twelve connection-types a Fracta can be in, and the links represent atomic actions a module can take. Fracta's have a novel way of initiating atomic movements discussed in sec. 2.3.2 using their connectors to repel them to or from other Fracta without the use of any degrees of freedom to reconfigure. The 'distance' between two connection types is

[12]Binary for Fracta; Up or Down

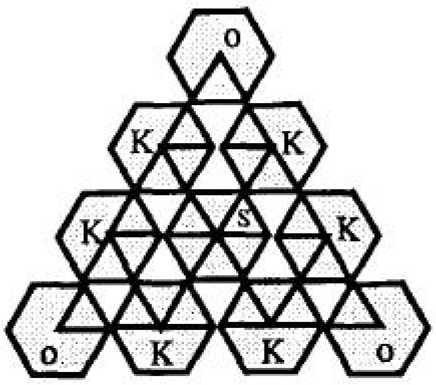


Figure 3.11: Example Fracta cluster, from Murata *et al.* [26]

the shortest path (least number of links) between them. Each Fracta can always check its connection type by polling all 6 connectors enquiring which is connected. It can also ask what connection-type each connected neighbor is.

3.4.2 Representation of the Fracta cluster

To describe the state of the modular cluster (which no one module can find out itself), the program lists all possible combinations of module connection types together with their connected neighbors' connection types. For example as seen in Fig. 3.11 shows a 10-Fracta cluster. All corner Fractas are of type '*o*' by a quick reference to Fig. 3.10a, and they are all connected to two 'K' type Fractas. All 'K' Fractas are connected to one corner (*o*), two other side Fracas (K) and one central Fracta (s). All together there only 3 of these combinations in this example (I shall refer to these as 'connection states':

o{K, K}
K{o, K, K, s}
s{K, K, K, K, K, K}

Where the LHS term is that of a Fracta's connector type, and the subset is the list of connected neighbors' connection types. This is how the configuration in 3.11 is represented by the planner.

3.4.3 Algorithm

Each Fracta module is programmed with the desired configuration of Fracta cluster, encoded as above, but will never know the current configuration of the cluster. So each Fracta must work with local knowledge only, to converge into this cluster configuration. One configuration example is shown in Fig. 3.12, a 10-module line shaped cluster attempts to change into a triangle. No Fracta's beginning in that line shape will know which position it will assume in the desired triangle configuration in advance, this is decided in due course.

A metric called *fitness* is introduced, acting as a heuristic each module can calculate for itself independently. This informs a module how far away it is from any one of triangle's 10 'connection-states'. It is mathematically defined by:

$$fitness(i) = \min_{j=1}^{M}[d(type^f(j), type(i)) + \sum_{k=1}^{6} d(ntype^f(j,k), ntype(i,k))]$$

where

M: number of final types

$d(a,b)$: distance between type 'a' and 'b'

$type^f(j)$: j-th final type

$type(i)$: type of i-th Fracta

$ntype(i,k)$: k-th term in the list of neighbor types of the i^th Fracta

$ntype^f(j,k)$: k-th term in the list of neighbor types of j^th final type

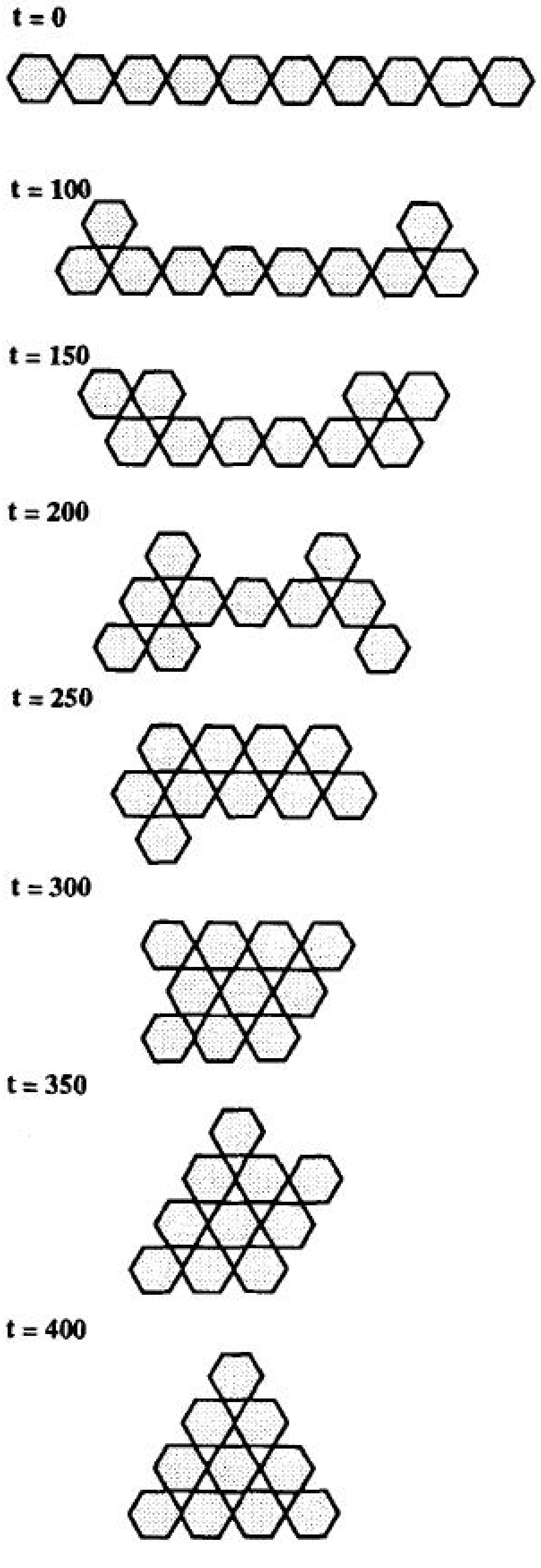


Figure 3.12: Simulated steps of a reconfiguration, from Murata *et al.* [26]

The ‘minimum’ function is in reference to whichever is the most similar of those 10 final connection-states to the Fracta’s own current connection-states. The first term is a difference measurement between the module’s current connection type and the most-similar final connection type, and the second term is to include the difference between the module’s neighbor’s current connection types and the most-similar final neighbor’s connection types. One module can compute this as it knows its current connection-states (only), and it knows every final connection-state. The lower the fitness function of a module the closer it is to a final connection-state. If the fitness is zero then a Fracta is in one of the final connection-states, if all Fracta’s have fitness zero then the cluster has successfully reconfigured.

Strategy

Using this fitness function, the designers of Fracta found a deterministic strategy of module movement based on a fitness-priority was difficult. Instead they opted for a less complex stochastic strategy: ‘A Fracta is allowed to move if fitness is above locally connected average, direction of movement is random’. So a module must first compare its fitness function with that of its neighbors. Under this policy, if a module’s random move decreases its fitness level, next time it compares to its neighbors fitness levels, it may be under the average and will stay where it is. I.e. moves that improve (decrease) a module’s fitness stay put, whereas moves that detriment (increase) a module’s fitness will continue to move, possibly just to reverse a ‘bad’ move it just preformed. In this way, individual modules are able to self assume, converging the entire cluster to the desired triangular configuration.

3.4.4 Experimental Results

The Fracta group tested this planning algorithm by running 1000 simulations of the configuration shown in Fig. 3.12. The results revealed 97.2% of trials reconfigured successfully before 2000 atomic movements occurred[13]. The remaining 2.8% either took longer to reconfigure or fell into reconfigurable stalemate; where no single Fracta module could perform a move[14].

The group's intuition reported that due to the stochastic nature of the process, convergence was slower for more modules, such as a 15-module line reconfiguring into a 15-module triangle. In this case 68.6% of trials reconfigured successfully before 4000 atomic movements.

3.4.5 Discussion

Based on these the results of both a 10-module and 15-module reconfigurations, the time complexity of such an algorithm seems to be quadratic-polynomial or greater. The way in which modules communicate strictly locally and the global cluster can converge to a desired configuration is impressive, and shows that such a simple self-assembling planner is possible.

However the way in which this planner represents cluster configurations by listing possible combinations of connection states (sec. 3.4.2) could cause problems for some desired

[13]listed as *time steps* in Fig. 3.12

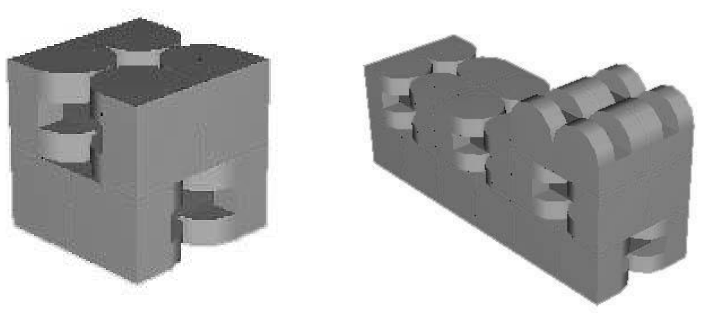
[14]Some connection types of Fracta are unable to move, such as 'm' in Fig. 3.10a. A hexagonal ring of Fracta modules composes 6 'm' connection-type Fractas, which is one instance of a reconfigurable-stalemate

configurations, because this method of encoding a configuration does not guarantee one-one mapping of code-to-configuration. The same code could represent multiple configurations. To the author's knowledge, the encoding of the triangle shape used in example Fig. 3.12, listed in sec. 3.4.2, is in fact unique by virtue of the triangle's symmetry, but more complex asymmetric shapes could not be expected to reconfigure successfully.

3.5 Case 2: MTRAN Planner (2002)

The designers of the MTRAN module developed two methods of reconfigurations specific to the MTRAN design [28]. The first method was a graphic user interface (GUI) which facilitated manual coding of a reconfiguration, allowing a person to enter in desired positions and configurations of modules at every reconfiguration step. The program gives helpful warnings such as if a module would be disconnected with such a move, or if a collision was imminent, the stability of the robot structure in a gravitation field, and also allows macros to be recorded, short sequences of movements that can be copied by other surface moving modules traveling to the same location [25]. The second method was an autonomous hierarchical planner, published in *A Self-Reconfigurable Modular Robot - Reconfiguration Planning and Experiments* [53], which this section examines. This autonomous planning algorithm imposed several configuration limitations in order to simplify the planning process, but nevertheless produced some impressive results with experiments preformed on real module hardware. No name was given to this algorithm, so it shall be referred to as the *MTRAN planner*.

A large configuration space is a major problem in reconfiguration planning, contributed by the many combinations of positions, orientations and rotations about degrees of freedom



(a) A block of four modules (b) A cluster of 3 module blocks and 2 converter modules

Figure 3.13: Compositional structures of modules of the MTRAN planner, from Yoshida *et al.* [53]

a module a can have. One way to combat this is to reduce the dimensionality of modules, such as the two dimensional module Fracta [26], or to develop an isotropic module, in which case there is no need to consider the configuration of a module, as all configurations are functionally identical. So an isotropic module is defined by position alone. An MTRAN module alone is geometrically non-isotropic, it has 54 possible configurations in 3D space[15], however the MTRAN planner restricts its planner to configurations composed of isotropic metamodules, 4-module blocks as seen in Fig. 3.13a composing of 2 different module orientations for each level. There are several advantages to this:

- Configuration space is reduced by factor 54, as MTRAN configurations are not considered
- By considering meta-module units and not individual module units, the number of units to consider is quartered, which is an exponential reduction of configuration

[15]Proof: MTRAN modules are a rectangular prism shape of which the length can point in 3 possible Cartesian directions. For any given direction, the module can be spun 90 degrees about its lengthwise vector resulting in 2 possible axle directions. Additionally both module parts are actuated and can be turned to 3 discrete angles (-90, 0 and 90 degrees). Thus $3\times2\times2(3) = 54$

space

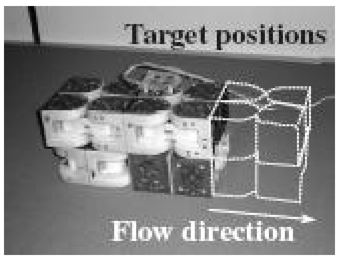
- Connector locations are ‘standardized’ over a modular cluster, and hence ‘predictable’, removing the need for a search algorithm to search for paths between connectors when motion-macros (Fig. 2.9) can be executed specific to this standardized surface.

- Connectivity between adjacent module blocks is guaranteed in any direction

3.5.1 Algorithm

The MTRAN planner solves reconfigurations by a hierarchical process of 2 parts; a global ‘flow planner’ and a local ‘motion scheme selector’ shown in Fig. 3.15. The global flow planner directs motions of module blocks for global robot movement. It selects the rear-most module blocks of robot and directs them the head of a robot with a set of optional paths module can take, for robot locomotion, resembling a ‘flowing’ action (Fig. 3.16). During this transition period, the 4 members of a module block separate and travel to the robot head serially. Once all there they re-form into their block, such that global planning of isotropic units can continue.

The local motion scheme selector takes the set of path options given to it by the global planner and assesses which is kinematically possible by decomposing into a set of local module motions allowed by the *rule database* (Fig. 3.15b). A valid path is one which the modular robot is always fully connected, and module collision is avoided. It searches through the set of given paths in order of shortest to longest path distance, executing the first valid path. The reason some impossible paths are passed in by the global planner is it is unaware of module design specifics and hence the module’s native kinematics.

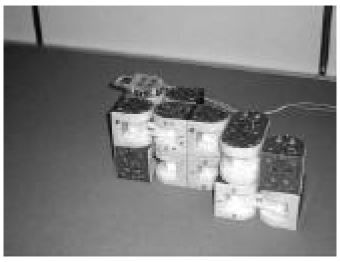


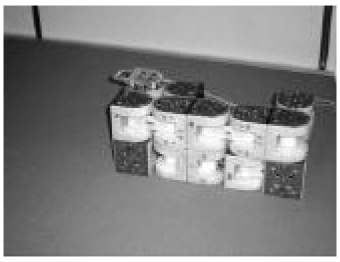
(a) Initial State

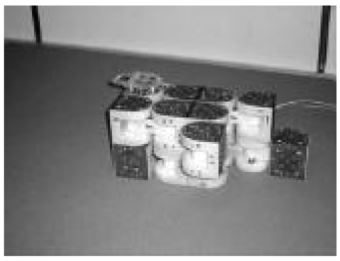
(b)

(c)

(d)


(e)


(f)


(g)

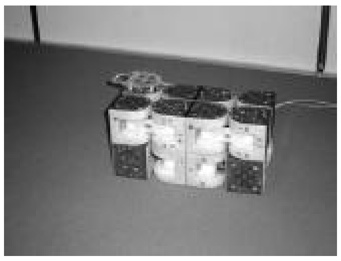
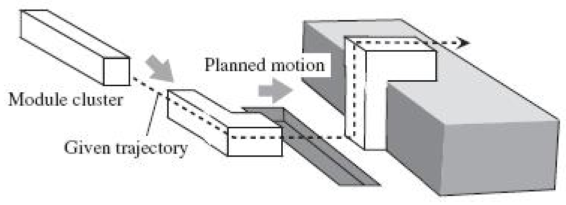
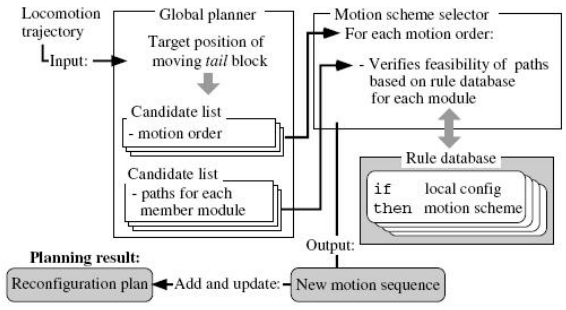

(h) Final State

Figure 3.14: Block (metamodule) Relocation of an MTRAN robot, from Yoshida *et al.* [53]



(a) Global ‘flow’ planner



(b) Concept Map

Figure 3.15: MTRAN Global/Local Reconfiguration Planner, from Yoshida *et al.* [53]

Though arbitrary reconfigurations are not possible for MTRAN robots entirely composed of these meta-module blocks owing to the problem of module *flavor* as seen in sec. 2.3.3, as one MTRAN module alone unable to reorient. So the MTRAN planner incorporates *converter* modules (Fig. 3.13b) mixed in with the metamodules which offer cooperative module movements (Fig. 2.9c) required for modules to change axle directions. These converters are required particularly at bends in the robot topology, where surface moving modules need to reorient (Fig. 3.16).

3.5.2 Discussion

The MTRAN planner's use of hierarchical search successfully shows how centralized deterministic planning can navigate a vast state space. Instead of considering the possible

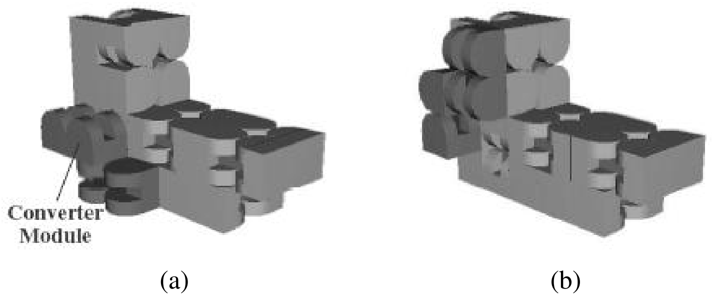


Figure 3.16: Direction change of a surface moving module from (a) to (b) via converter cooperation, from Yoshida *et al.* [53]

movements of any module at any time (which would be incredibly inefficient for a centralized planner) the algorithm focuses on the relocation of one module at a time because of the division of *global* and *local* searches. This division is a classic example of *divide and conquer*, limiting task complexity into manageable sequences of relocation tasks.

Whilst the choice of restricting configurations to metamodule structures does reduce the state space size significantly, the robot is limited to more forming more 'pixelated' or 'blocky' shapes. It does not have the capacity to form finer geometrical features that involve single module widths which could possible restrict its usefulness as a robot.

3.6 Case 3: Claytronics Planner (2006)

The Claytronics planner [36] and module have been a continual development from Carnegie Melon University since 2004, publishing many papers [11] refining a planning technique inherent to their Claytronics module (Fig. 3.17). Their paper titled *Scalable Shape Sculpting via Hole Motion: Motion Planning in Lattice-Constrained Module Robots* [36] in 2006 was their first publication specified key details about how their program was implemented, which this section focuses on. Since then the research group has fully decentralized their

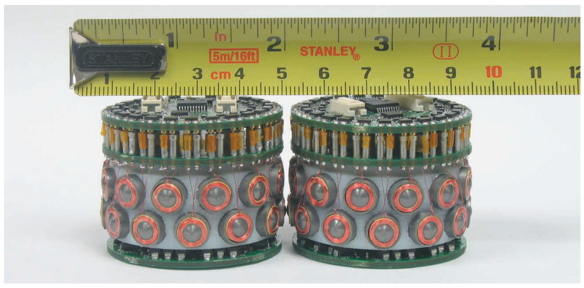


Figure 3.17: Claytronic module prototypes (2005), from Goldstein *et al.* [19]

implementation and developed two "modular-robot-specific declarative programming languages, *Meld* and *LDP*" they argue is much better suited to modular robotic computer architectures and provide a an abstraction to effectively execute a single program over a group of modules considering module-module communication can have significant time delays [2, 37].

Claytronic modules (Fig. 3.17) are cylindrical and will pack into two-dimensional hexagonal lattice patterns. Like Fracta, they have no degrees of freedom, and use magnets for locomotion. The reconfiguration planner uses the motion of *holes* rather than specific module movements, inspired from semiconductor physics [19, p.101], in order to reconfigure. A hole is a cavity within the modular cluster as shown in Fig. 3.18, whereby 7 modules are 'missing' forming a hexagonal hole[16]. The 12 modules that line a hole are referred to as the *shepherd* modules. In order for a hole to move, the three shepherd modules closest to the white arrow in Fig. 3.18 relocate themselves to the opposite side of the hole. This moves the holes center one module diameter in the direction of the arrow.

To reconfigure, a planner can either create or destroy holes along the surface or the cluster, thereby expanding or contracting the cluster surface at a particular edge. For example, the creation of a hole can be seen in Fig. 3.19, whereby surface modules form a shepherd

[16]holes are always 7-module hexagonal shapes

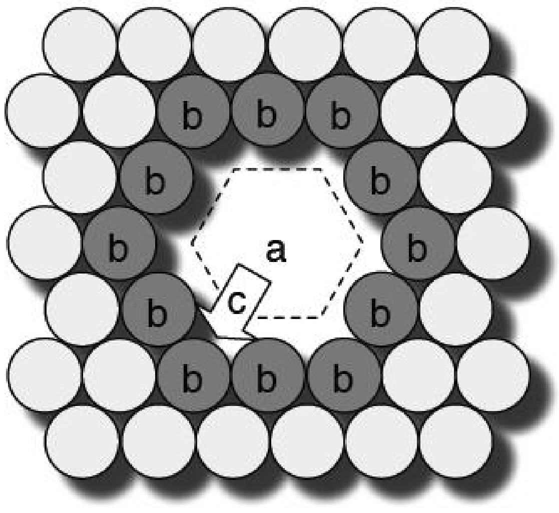
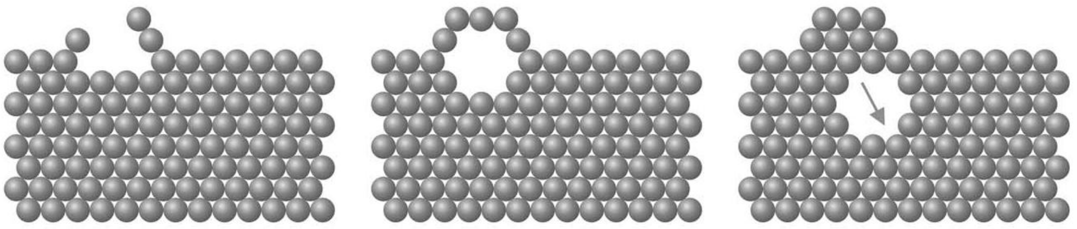


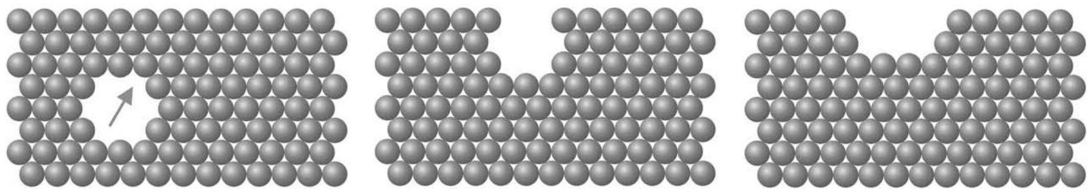
Figure 3.18: Hole basics: a) hole, b) shepherd modules (dark), c) direction of motion, from De~Rosa *et al.* [36]



(a) An edge selected to expand, (b) creates one or more holes, (c) launching them into the ensemble

Figure 3.19: Expanding edge by creation of a hole, from Goldstein *et al.* [19]

group of 12 and thus create a new hole in Fig. 3.19b. From here, the hole can move away into the structure, leaving the previously flat surface locally elevated. The opposite can happen too, whereby a hole is destroyed in order to contract a surface, as seen in Fig. 3.20. As a hole approaches the surface of a cluster, it can simply continue until its ‘emptiness’ is consumed by the exterior, leaving a local gouge in a surface. By creating or destroying many such holes along surfaces, a robot either can expand or contract regions of its body in order to reconfigure.



(a) A hole approaches a contracting edge, (b) but rather than reflecting, (c) the edge consumes the hole

Figure 3.20: Contracting edge by deleting a hole, from Goldstein *et al.* [19]

3.6.1 Algorithm

The Claytronic planner is a decentralized one, modules to act upon local knowledge only, and thus the program is very scalable. To implement this, global communications between modules is not allowed, and so hole motions are not coordinated in any way, they travel randomly within the structure bouncing of exterior surfaces if not needs much like molecules of an ideal gas. Through randomization of direction, the global density of the structure will always remain roughly balanced, so the robot will (most likely) not loose balance unpredictably.

The definition of 'local' above is determined by the size of *tri*-regions seen in Fig. 3.21. These regions are the resolution of new shapes the robot can morph into, and modules within will communicate to each other and share local computational tasks like, hole creation/deletion. Each Tri-regions mark themselves either for deletion, filling, or to remain unchanged depending upon whether they exists in the goal configuration or not. So as one tri-region along the robot surface fills up its region with modules by creating holes, the holes' which move randomly will eventually arrive to a tri-region attempting to delete itself, which will use those holes to do so by destroying them at its surface.

In this way reconfigurations are possible, yet there is still the potential for configuration

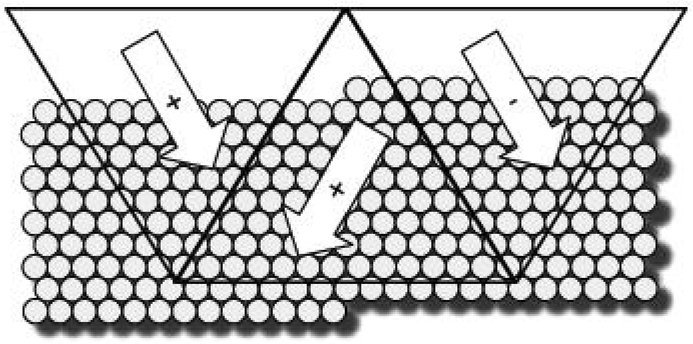


Figure 3.21: Tri-regions: left and centers *tris* are set for growth, the rightmost for deletion, from De˜Rosa *et al.* [36]

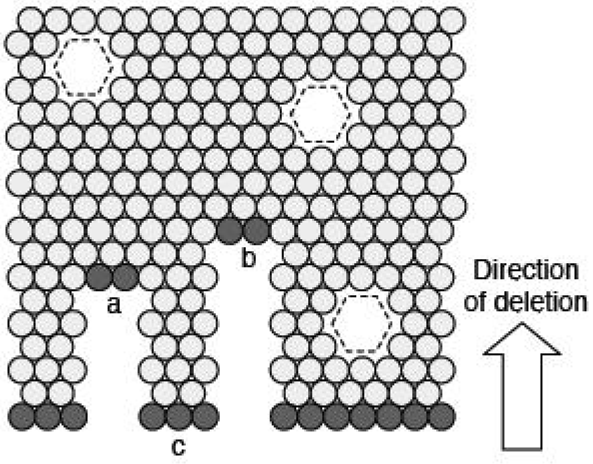


Figure 3.22: Starvation example in absence of smoothing effects. Regions a and b prevent region c from forming a hole to delete itself, from De˜Rosa *et al.* [36]

stalemates. Fig. 3.22 shows an example of a local stalemate whereby previous hollowing regions 'a' and 'b' continuously resulted in a region of modules 'c' which is not too slim to form holes, and thus cannot remove itself. So a method for enforcing uniform hole destruction along a collapsing surface is required. The planner does this by prioritizing the most extreme protrusions of a surface for deletion over already hollowed regions. It also borrows concepts form of simulated annealing, whereby hole destruction at a surface is probabilistically dependent on distance between the current surface location and that of the 'corresponding' surface of the goal configuration. As holes travel, their lifetime is also bound by a logarithmic time decay value which increases their chance of destruction at a surface the longer they live. The probability of hole destruction is called the *Temperature test*:

$$p = 1.0 + \log \frac{d_{near}}{max(d_{max} - c_{decay} \times t, 1)}$$

where

d_{near}: distance (in diameters) between r and the closest point on the target geometrys perimeter

d_{max}: maximum distance between the perimeters of the source and target geometries

c_{decay}: time decay constant (typically 15 to 50)

t: time elapsed (unit of *time steps*; defined by the time for a hole to move one module spacing)

The temperature test takes care of these more macroscopic sources of local configuration stalemate, however another consideration on a more microscopic level the planner must consider is surface *roughness*. For hole to be created or destroyed, they require fairly flat surfaces to do so. One hole being created or destroyed will increase surface local roughness (as seen in Figs. 3.19 & 3.20) which can restrict further hole creation/destruction until smoothed. Tri regions smooth task responsibility for their own surface smoothing by choosing a direction away from the robots exterior (arrows in Fig. 3.21) to implement 'gravity driven collapse'. Much like sand does naturally, gravity driven collapse coordinates modules of higher elevation to move 'down' to fill local depreciations.

3.6.2 Experimental Results

Reconfiguration experiments were carried out between the fours shapes shown in Fig. 3.23. These shapes were chosen to test the planner's ability to create and remove both shape corners and curvatures. These included morphing:

1. The square shape into the 'T',
2. The 'T' shape back into the square, and
3. The rectangle into the circle

Module numbers were varied, such that the square would be trailed with side sizes 100, 200, 300 and 400 module lengths, so there were 12 experiments all up, each repeated 10 times, with random hole placement and motion directions each time, running for 10,000 time steps. The number of tri regions was fixed over all 12 experiments.

Several measured of performance were recorded for each time step of the reconfiguration, perhaps the most meaningful was the *shape compliance fraction* which is the percentage of modules already in the goal shape. Fig. 3.24 shows the reconfiguration progression

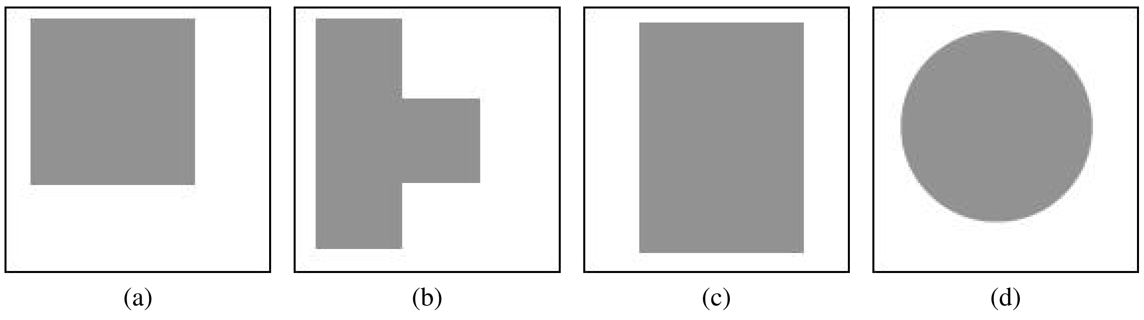


Figure 3.23: Experiments performed on reconfigurations between these shapes, from De˜Rosa *et al.* [36]

of all 12 experiments over the time in terms of this measure. Besides showing that bigger shapes do indeed take more time to reconfigure, that the square to 'T' experiment was slower than the other 2 experiments, and caused global disconnections to occur. The designers of Claytronic attributed this to:

1. Initial motion completeness for this experiment began less than the rectangle to circle experiment
2. Increasing module numbers for a fixed set of tri regions causing increased in coarseness and modules to control which led to a "decrease in the control of local curvature" which was a more pronounced effect from the square to many-cornered 'T' many-module experiment

3.6.3 Discussion

It is unclear whether any of the above experiments eventually converged to the goal shape, perhaps only not due to tri region resolution. Enforcing global connectivity would need to be addressed next, as separating into two distinct modular clusters does not guarantee a

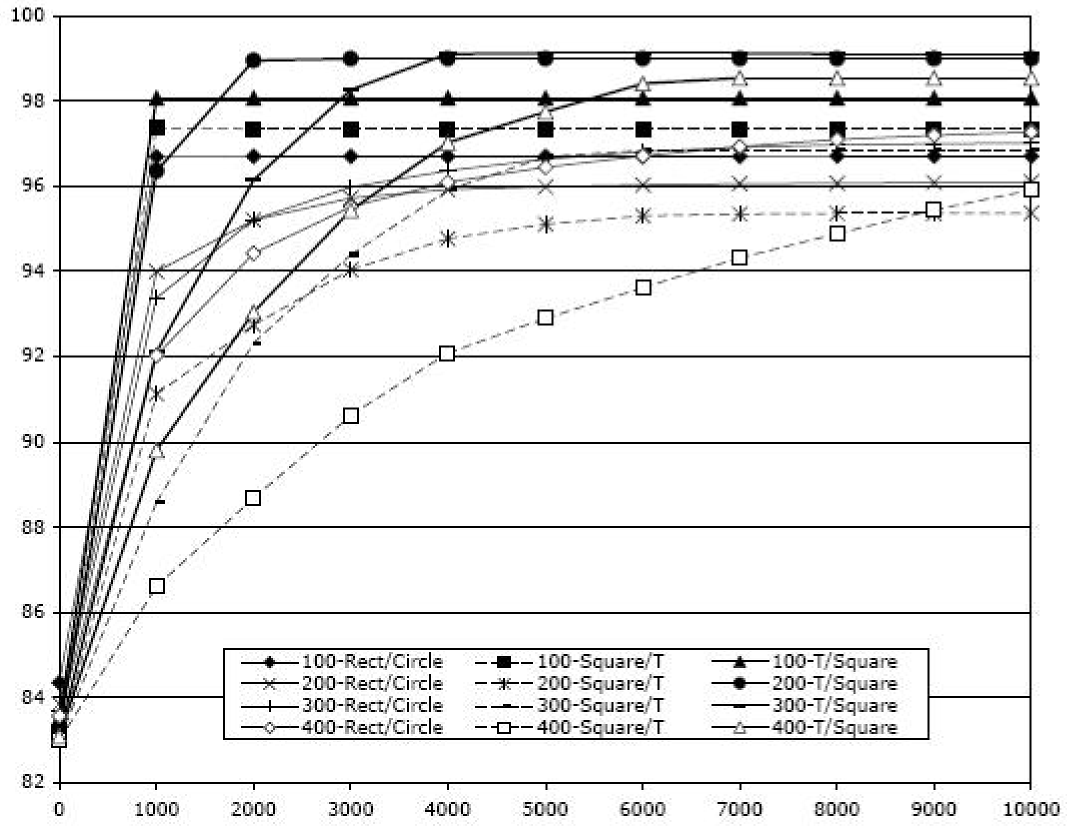


Figure 3.24: Shape compliance of Claytronic experiments over time, from De˜Rosa *et al.* [36]

robot can ever reconnect. It suggests from the square to 'T' shape experiment that highly convex shapes are potentially unsolvable, or incur significant delay to converge to the given shape. One also wonders if holes can become bottlenecked at slim places between two regions, and additionally if their passing at bottle necks with considerable effect the structural integrity there.

However Claytronics planner's innovative use of *holes* to change shape is extremely impressive and presents a Claytronic robot with some interesting robot abilities most other planners can't provide. For one, hollows within a structure can be easily formed is so desired. Individual modules inside the structure are never 'stuck' either, they can always use holes to travel themselves, much from freely than seen in the ATRON example (sec. 2.3.3). Some planners, like Million Module March (sec.3.7) if planning to construct a shape with a cavity can accidentally leave a module in there, which when called later to move is unable to do so. In this case a hole could be used to absorb the module out. Perhaps this is the only use of internal module movement in a homogeneous Claytronic robot, but if a heterogeneous robot has a special sensor say module buried deep within the structure, it will be easily retrievable when its use is required on the surface. Exact routines to control these advantages have (assumedly) not been written into the Claytronic planner yet, but its method of reconfiguration would allow for their realization.

3.7 Case 4: Million Module March (2008)

Million Module March [14] in essence is a hierarchical search with two major levels similar to the MTRAN planner. The high-level search routine is concerned with selecting & instructing certain surface cubes that are allowed to detach from the rest of the structure

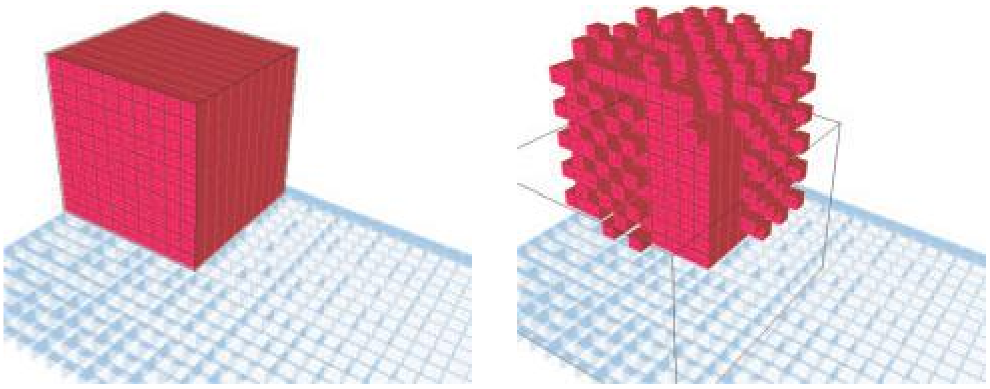

(a) Initial configuration (b) One time step later

Figure 3.25: A Million Module March robot, initiating a reposition, from Fitch and Butler [14]

to become mobile and move towards a particular location. This instruction is called as a subroutine, which is the lower level part of the program's search hierarchy. This lower level search is required is to work out the details of what path a cube can move over the structure to its given goal location.

This algorithm was developed with the intention of locomotion of a modular robot around obstacles to a chosen goal region, and without the overhead of reconfiguring into exact shapes, it can execute and entire repositioning of the robot in sub-linear time with the amount of module it composes. This makes the program extremely scalable, and has thus been able to simulated repositions of robots up to 2.2 million modules large. It used constant memory per module, and localized communications between modules to take advantage of highly parallel movements without global synchronization as shown in Fig. 3.25.

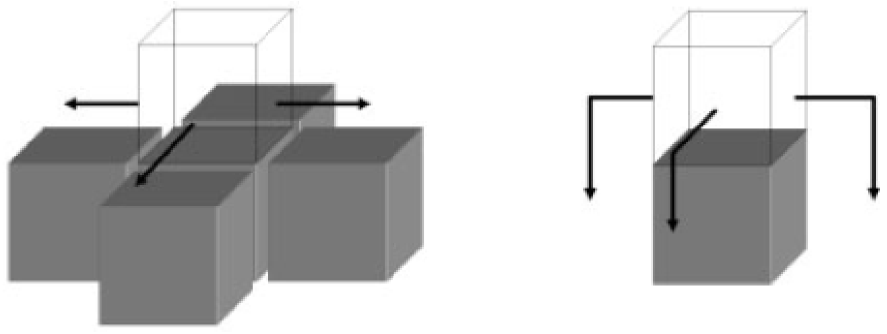

(a) Sliding transitions (b) Convex transitions

Figure 3.26: Sliding-cube movements possible, from Fitch and Butler [14]

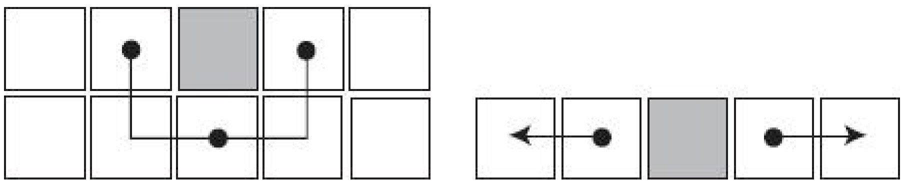
3.7.1 Cubic Metamodules

The *sliding cube* modules used in this planner represent general metamodules, made from any particular type of module. Not all module types can form into cubic meta-modules, however there are still elements of this planner that can be taken advantage of, discussed next. Fig. 3.26 shows how sliding cubes atomic movements possible, either a sliding translate to an adjacent cubic cell, or turning a corner to connect to another face of a cluster-module.

3.7.2 Algorithm

Maintaining Connectivity

This program first begins with a connectivity check of the surface elements, identifying the cubes which can be safely detached from the structure without the structure becoming globally disconnected (sec. 3.1.3). Such modules will not detach themselves, but can safely become mobile, moving over the structure to a new location. To check their reposition does not violate global connectivity, they check that all their initially connected neighbors



(a) A dense cluster; connecting cycle links neighbors cubes via other cubes

(b) A chain of cubes; a connecting cycle does not exist

Figure 3.27: Connecting cycles, from Fitch and Butler [14]

are connected to each other independently from them, via a *connection cycle*. For example, Fig. 3.27a shows a grayed cube initiating a connection cycle check on its neighbors, marked with black dots. A connection cycle exists along the black lines connecting those 3 neighbors, so if the grey cube relocates then these cubes are still all connected to each other. In this way global connectivity is guaranteed before the grey cube moves. Fig. 3.27b shows a connectivity search initiated in a chain of modules, the iterative-deepening search progress outwards either side and does not find a connection cycle. In this case the grey module cannot move, as it would separate the two leftmost modules from the two rightmost modules. If a connection cycle exists, a module will lock it those immediate neighbors and the position it intends to move to for the time it takes to move to prevent collisions with other modules.

Navigation Function

With a known set of these modules which are allowed to detach and become mobile via connectivity checks, Million Module March uses aspects of reinforcement learning, particularly dynamic programming (appendix.3.2.3) to propagate a navigation function out from a goal region, to assign every surface connector on the robot a value of desirability for any surface moving modules to consider. The mobile modules are then programmed to each greedily select ever more desirable connectors to swap to until they are inside a goal region, indicated by a wireframe box in Fig. 3.25b. The net result is a mass of cubic modules, with a surface that is ever-flowing towards the goal region which inches the entire robot into the wireframe.

This propagation begins with any modules that have connectable faces that correspond to cell positions within the goal region[17], they will be given desirability values of '0'. At this point, surrounding cubes will repeatedly enquire as to what their value of desirability is, and copy that value for themselves with the addition of a cost value of '-1' to represent the time & energy a surface moving module would have to expend in transitioning both connecting faces. This process will continue outwards from the goal, with the value of successive connectable faced decreasing by '-1' of their closer neighbors. Sometimes a face may have multiple neighbors with different values, in this case a connecting faces will choose the highest value. Say a connecting face has two neighbors of values '-3' and'-7'; it will gives itself a value of '-4' (by choosing the neighboring value of '-3' and adding a cost of '-1' to account for a module's transition). This process only relies on local communication between modules, which allows the algorithm to scale well.

When propagated over the entire robot, connecting faces can 'point' the way to more desirable connectors (highest value) for surface moving modules to follow, defining optimum paths towards the goal region exampled in Fig. 3.28. As many modules are in motion at any one time, some can block the path of others and so the navigation function needs to be continually updated (by continual propagation) so a surface moving module can re-plan its motion path if need be.

[17]goal regions are always placed next to initial robot regions, so it is guaranteed that some modules will have connectable faces which correspond to lattice-cell positions within the goal region. If a robot is to locomote further away, a series of goal regions can be used

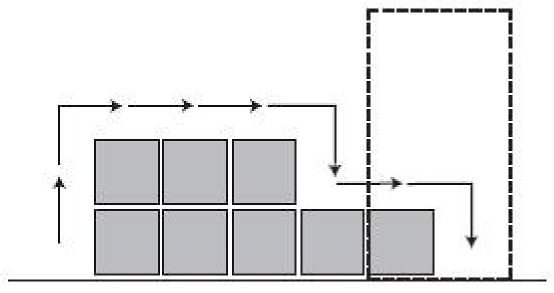


Figure 3.28: Navigation function pointing the direction of an optimal path for surface moving module to travel into the goal region, from Fitch and Butler [14]

3.7.3 Discussion

Million Module March provides an effective means of coordinating many surface moving modules, replanning transition paths if conflicts occur, to reposition a modular robot. This navigation function labeled 3D cell positions with values of desirability (the cells that correspond to available connectable faces). Such a searchable space is very small in comparison to other configuration planners, and brings the searchable space back into the realm of what is possible for regular search routines to plan module motions. Even though many module designs cannot form a cubic meta module in which to take full advantage of this planners benefits, the concept of this navigation function based on dynamic planning can still cross over to other robots. The searchable space would not necessarily by positions in 3D space anymore, more likely some form of state space depending on the module, but nonetheless can prove effective for a planner. Million Module March's sub-linear execution time shows that locomotion of large modular robots is possible and provides a good benchmark for future reconfiguration planner to aim for in the time-complexity of their reconfigurations.

3.8 Case 5: Graph Signature (2008)

Graph signature [1] is a stochastic optimization method that uses aspects of graph theory to discover optimal reconfiguration paths. By considering a modular robot as a graphical representation of part positions with directional edges for connecter genders, *graph edit-distance* can be used as a heuristic function to guide a search.

Graph edit-distance is a similarity metric in graph theory. It gives a numerical score of similarity between two graphs, based on the Maximum Common Sub-graph (MCS) and the shortest sequence of edit operations[18]. By representing a robot's initial and final configurations as configuration-graphs, this function directs the reconfiguration planner by assigning greater probabilistic weighting to actions that result in greater similarity with the final configuration.

Possible search paths are depicted in Fig. 3.29, which are a sequence of configurations shown by their similarity metric (vertical axis) as they converge on the final configuration. As shown, an optimal reconfiguration path may be beset with local optimums which would trap any greedy routines. Although the stochastic nature of the search here supplies the planner with an exploratory element which to ultimately get around this, it may require multiple searches until this solution is found.

In order to recognize a search thread stuck in a loop, or a new thread attempting to re-do old work, each thread keeps a log of configuration graphs previously encountered using a *graph signature*, a isomorphism-invariant code (hash code) of graphs. A thread is forbidden to re-visit previous configurations if presented the opportunity, and a new thread can be spared computations previously done.

[18]i.e. deletion & insertion of edges or vertices, that transform an initial graph to a final graph

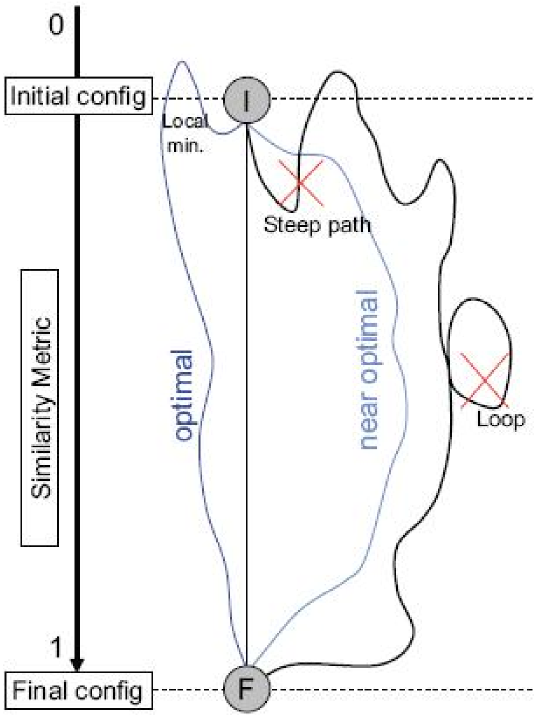


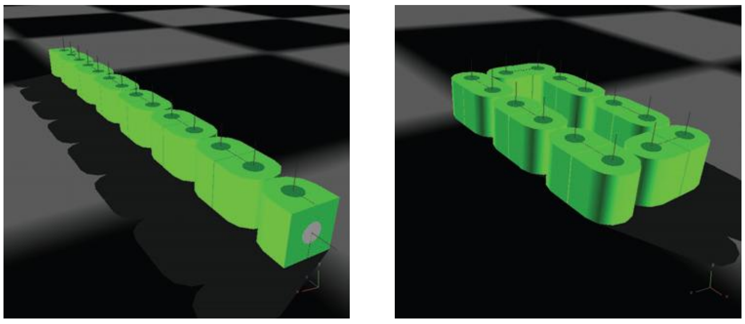
Figure 3.29: Searches using a similarity metric, from Asadpour *et al.* [1]

New search threads are run until the programs finds a “suitable solution”, of which is not further described [1]. The advantage of this program is its capacity to discover optimal reconfigurations, however this can take many searches to do so, and is not guaranteed to do so within any set time frame for real-time computing applications. Also there is no way of knowing when an optimal reconfiguration solution has been found, only that it is the ‘best’ solution found so far, so searching is not necessarily terminated at this point.

Some simulated experimental results were conducted with this algorithm using the MTRAN design. A configuration from a line to a ring configuration (Fig. 3.30) was chosen to test the relationship between a program’s execution time and the amount of modules present. One program action to another configuration is defined by one connection/detachment event, which could involve many module actuation events. The test of a line into a ring therefore constitutes one *action* which is the smallest iteration step this program’s search thread will take. Searching disconnection options is a quick exercise by considering existing connections, but searching new connection possibilities from the result of multiple module actuation combinations is an inverse kinematic problem and grows exponentially with degrees of freedom added. Fig. 3.31 shows this exponential relationship as the modules increase from 5-8[19].

Another similar experiment was conducted on the reconfiguration of a 4 modules from a quadruped into a line (Fig. 3.32). Conducting 500 separate search trials, each using 20 search threads, some statistical performance data was complied. The optimal reconfiguration sequence for this is 9 attach/detach actions. As seen in Fig. 3.33, never in the 500 experiments did the first search thread fins the optimal solution. This is indicative of a

[19]The scaling disparity between the 4-module point and the 5-8 module point in Fig. 3.31 is reported to mark the switch of their computer to using virtual memory, as main memory filled up)



(a) Initial line configuration (b) Final ring configuration

Figure 3.30: Reconfiguration Experiment Two, from Asadpour *et al.* [1]

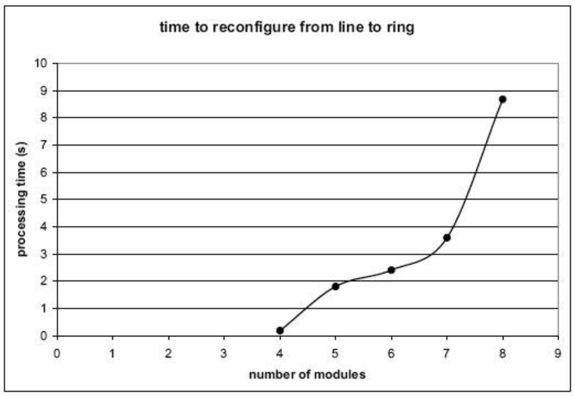
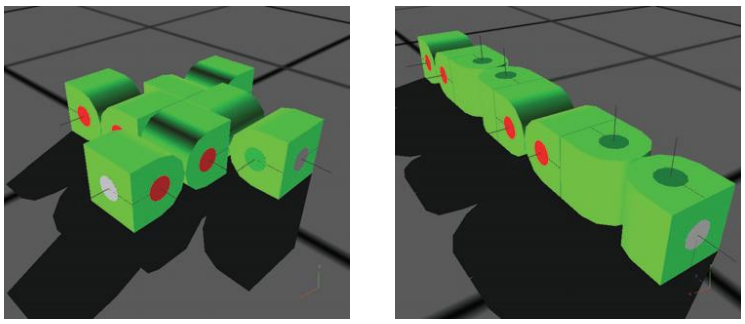


Figure 3.31: Computation time of a reconfiguration from a line to a ring, from Asadpour *et al.* [1]



(a) Initial quadruped configuration

(b) Final line configuration

Figure 3.32: Reconfiguration Experiment One, from Asadpour *et al.* [1]

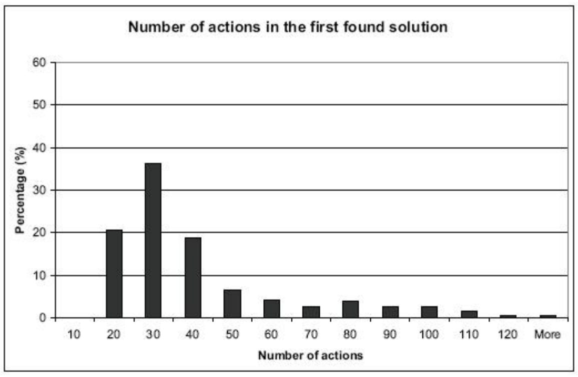


Figure 3.33: Computation time of a reconfiguration from a line to a ring, from Asadpour *et al.* [1]

local optimum misleading the heuristic driven search as seen in Fig. 3.29. However when considering the results of all 20 threads in Fig. 3.34 is appear a small percentage of the 500 experiments did find the optimal solution, but the far majority of 'best solution found' over 20 threads lie in the 20-30 attach/detach actions column.

In other statics, not shown figuratively here, 53% of the experiments found at least one reconfiguration solution after searching 50k configuration graphs, 83% found a solution after 100k graphs were searched. Understandably, more configuration graphs required searching on average before a 'best solution' was found out of the 20 search threads.

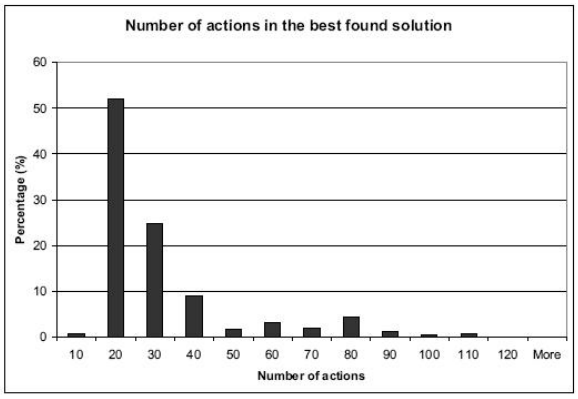


Figure 3.34: Computation time of a reconfiguration from a line to a ring, from Asadpour *et al.* [1]

3.9 Conclusions

As seen, SRP development can be a difficult task, there are many considerations and in all cases, SRPs cannot guarantee to solve reconfigurations optimally an also execute in real-time. This is due to the inherent challenge of a vast configuration space a planner must search through efficiently. The key to a useful planner is *scalability*; ideally a planner should be able to execute reconfigurations for an SRR of an arbitrary number of modules and not be restricted by an upper threshold of them. For this to happen, both time complexity and space complexity of an SRP execution need to relate linearly at most with the amount of modules present. Additionally an SRP must always be responsible for maintaining global connectivity and preventing collisions of modules, and to perform efficiently must have an intimate understanding of the module hardware it is planning for.

To combat the problem of vast configuration spaces before actual planning begins, often compromises can be made. One example is the use of metamodules which both the MTRAN and Claytronics planners use. Metamodules increase the isotropy of collections of modules (or lack thereof in the Claytronic's case) and thus reduce the amount of possible states these modules can be in for a given planner to consider. This reduces the search space size exponentially, though the downside is the set of possible robot configurations it can morph is limited to more 'pixelated' shapes, conceivably limiting the general utility of the SRR.

Good heuristic functions are not easy to develop. The designers of MTRAN found that using lattice spacing for anisotropic 3D modules almost gave no indication of the distance between two robot-configuration points in the SRR's configuration space. Different heuristics are often tailored to specific module designs and some cannot always warn of a configurable stalemate as seen in the Fracta experiments. Heuristics functions, or equivalent directing methods, are required for SRPs to be able to successfully search the vast configuration space in reasonable time. Yet often SRP designers do not consider this enough and commonly opt for hierarchical methods to make sense of the large space, or resort to strictly decentralized planning where module-module communication is restricted between modules only a few module lengths apart. In this way the configuration space does not have to be explicitly searched in a fully comprehended sense by the SRR. Instead self assembling behavior policies adopted by individual modules result in interacting behavior that (hopefully) converges SRR structures to goal configurations, completely unbeknownst to any one module (Fracta, Claytronics).

Self assembling and strictly decentralized module behavior is characteristic of stochastic planners, though not a definite indication that a planner is stochastic. The downside of these strictly decentralized planners is they are often unable to foresee either local minima (if the planning approach is gradient based [52]) or configurable stalemates that loom ahead. SRPs that aren't strictly decentralized, such as MMM, have the ability to apply to execute pseudo-global based routines like a connectivity-checker, that will search out through however many (though bounded by a maximum search depth) modules to see if a connection cycle exists. In this way, the planner can still reap the benefits of being decentralized (sec. 3.3.4) yet can also foresee certain disastrous events like global disconnections that the Claytronic planner for example cannot always foresee.

Of all the SRPs discussed the MMM by Fitch and Butler (2008) is the most scalable, and is also the most scalable algorithm that exists in the field of SRR. Its efficient use of a navigation function allows the highly parallelized motions of surface moving modules. As discussed, MMM is also able to foresee global disconnection events. For these reasons it was decided to utilize the navigation function and connectivity checker of the metamodule-based MMM, and re-implement them into native kinematic space in the design of this thesis.

Chapter 4

Centralized Planning for 3R-Type Modules

This material covered thus far is intended to give the reader the necessary background information about self-reconfiguration planning. This chapter presents a centralized implementation of a new SRP which solves arbitrary SRR reconfigurations of a homogeneous composition of the 3R module types[1] such as Superbot (sec. 2.3.5). This is a centralized serial reconfiguration planner written in Java is dubbed the 'Centralized Self Reconfiguration Planner' (CSRP) and is followed by a decentralized implementation (DSRP) presented in the next chapter.

This chapter commences with the problem definition and general approach taken. This progresses to a description of geometrical properties of the 3R modules and a discussion on *state representation*; how the CSRP represents any state a 3R module could potentially be in. Some of the specialized motion primitives are then discussed and follows with an

[1]*3R* is shorthand for 3-Rotation, a module with 3 degrees of freedom

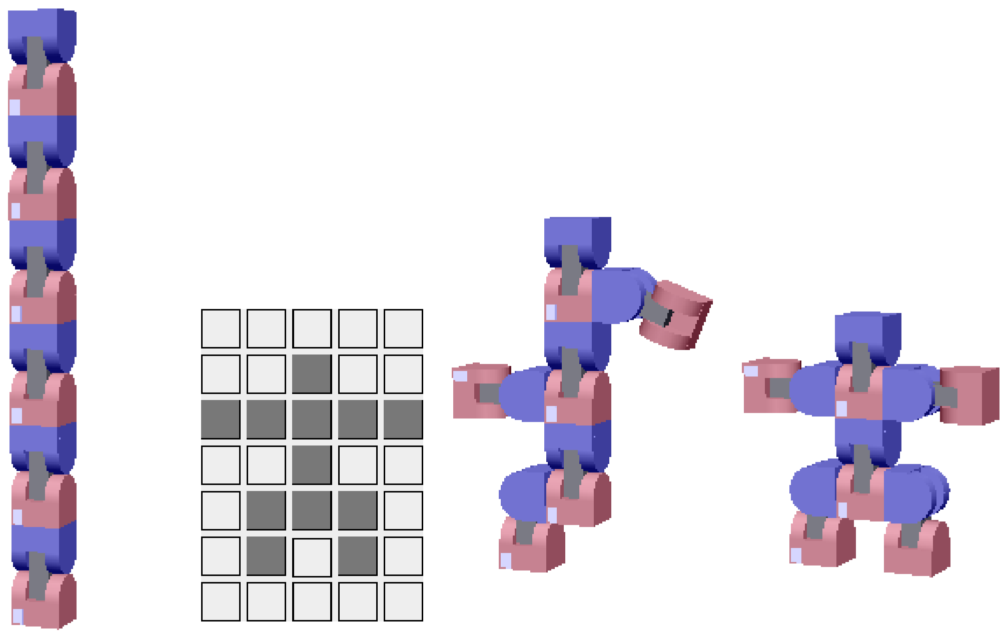
outline of the CSRP algorithm; all the routines it composes and how they are organized. An evaluation of the CSRP is given, including an analysis and several reconfiguration examples the CSRP is capable of. Finally a discussion concludes this chapter outlining the performance of this planner in the context of the literature and highlighting what has been achieved.

4.1 Problem Definition & General Approach

The task of self-reconfiguration is defined as follows: Given some modules that are interconnected in some way, find a partially[2] ordered sequence of actions that changes the modules from their current configuration into a desired one. A valid reconfiguration includes the avoidance of any module collisions and does not violate global connectivity (sec. 3.1.3).

The CSRP approach is is to move modules in serial (one at a time) to change a robot's configuration. An example is seen in figure 4.1a which shows a Superbot-based SRR of 6 modules in an initial shape. A graphical user interface (GUI) is used to input a desired shape the robot should transform into (Fig. 4.1b). This GUI is purely for testing purposes of the CSRP, because in a fully autonomous SRR a higher level program(s) will use this SRP to change shape instead of a human operator. The CSRP then begins searching out the best reconfigurable solution which Fig. 4.1c shows a snapshot of mid-execution in simulation. In this snapshot only the rightmost module is currently moving, traveling downwards. Finally the robot converges on the desired shape of Fig. 4.1d.

[2]some modules can move in parallel



(a) Initial configuration (b) A desired shape is inputted with a GUI (c) SRR in mid-reconfiguration (d) Final configuration

Figure 4.1: A line of six 3R modules reconfiguring into a stick figure

The CSRP uses a hierarchical search as did MTRAN and Million Module March (sec. 3.5 & 3.7). The advantage of this method is that it embraces a ‘divide and conquer’ advantage that makes it feasible for a deterministic planner to navigate a large state space. The two levels of search are the global and local searches. The global routinely selects modules for mobilization; to relocate from their current positions to goal locations. The local search determines optimal motion paths between those locations according to the module’s native kinematics (sec. 3.1.1). The local search does this by building a *state network* for each mobilized module; a network of possible *states* a module can possibly assume whilst all other modules remain static. These states are linked by valid[3] *actions* the module can perform respective to each state. The interplay of *states* and *actions* in this design is very much similar to that discussed throughout the reinforcement learning section 3.2. All actions change a module’s state, and the state before and after an action are *linked* by such an action. A visual representation of such a network is shown in Fig. 4.2 where each circle represent a state-node a module can assume, and each link represents a valid action performable from that state which will lead to another state-node. For the local search to relocate a module, it must find a (preferably short) path that exists in this network that goes from state-A to state-B. If the SRR then executes those module-actions defined by the solution path, it will reconfigure correctly into the desired shape. The CSRP creates a state network for each module relocation (iteration of global search), however the DSRP takes advantage of re-using such a network for multiple modules to utilize, presented in chapter 5.

The advantage of this planner is it operates in native kinematic space. It is therefore able to take full advantage of the module’s hardware, by defining possible module movements as

[3]*valid* means taking such an action is not outside the physical capabilities of the module (when Superbot part has rotated to the maximum position of its 180 degrees range it cannot turn forwards anymore) and also such an action would not cause a collision

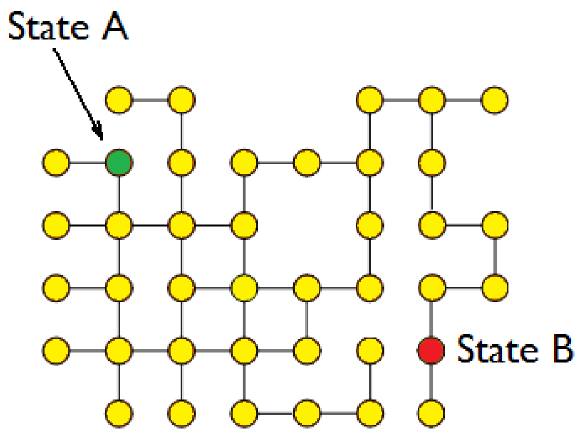


Figure 4.2: State-Network Visualization

to exactly what is allowed and what is not supported by the hardware without making any assumptions. Therefore module relocations are guaranteed to find optimal transition paths, and so Local Search is both a *complete*[4] and *correct*[5]. Additionally this planner does not use metamodules. Even though metamodules can help reduce the state space size, the CSRP can still find solutions independent of this advantage and thus does not compromise robot versatility and application by restricting operation to a subset of possible configurations. The CSRP is also deterministic; for each reconfiguration is executes, it guarantees it can always re-execute the same reconfiguration in the future if the need arises. Furthermore it is able to re-execute any reconfiguration in the exact amount of time it took to reconfigure in the first place. Stochastic SRPs such as Fracta, Claytronics and Graph Signature (sec. 3.4, 3.6 & 3.8) cannot guarantee this (except in a probabilistic[6] sense).

Full generalization of the CSRP has not been holistically attempted yet in the face of more fundamental challenges such as implementing cooperative motion control between

[4]definition: if there is a solution it will find one

[5]definition: a solution which it finds is guaranteed to be a correct solution

[6]an example of a probabilistic guarantee is 'being able to successfully reconfigure from shape-A To shape-B 90% of the time', or 'can reconfigure between two shapes within 20 seconds, 65% of the time etc.'

interdependent modules (*helpers*; sec. 4.4.1). Challenges of parallelization and decentralization have been successfully implemented in the DSRP, discussed in the following chapter. Thus generalization is the sole challenge left as per the ultimate goal of this thesis '*To create a decentralized program that can autonomously plan arbitrary & parallel reconfigurations of homogeneous modular robots of arbitrary module designs in native kinematic space*', and is left for future consideration (sec. 6.3.3). As the CSRP is not fully generalized, the 3R module was chosen to instantiate its implementation due to the module's versatility. The 3R module has a high number of both degrees-of-freedom (3) and connectable faces (6). Additionally it does not suffer from the *flavor* phenomenon (sec. 2.3.3) like ATRON and MTRAN do.

4.1.1 Terminology

Before this chapter progresses into CSRP specifics, the different classes of modules need to be defined. The CSRP brands each module in its SRR as one of the following classes (with associated roles):

- **Static Module:** A module that does not move. It makes up part of the robot's structure
- **Mobile Module:** Also *Surface Moving Modules*, A mobilized module traveling over the surface of static modules which form the bulk of a modular cluster
- **Helper Module:** A module that has one static part and one free to aid mobile modules move by picking them up at one location and placing it down in another, without detaching itself from the module cluster

Additionally the term *Structural Module* refers to a module is either a *static* module or a *helper* module. It is the 'opposite' of a *mobile* module. During a reconfiguration some modules will change class. If a structural module is selected to become a mobile module, it is said to have been *mobilized* and is thus able to move along the robot's surface of structural modules.

4.2 State Space Reduction

There are two exploitations of hybrid modules the CSRP utilizes. One is a hybrid's ability to act as a purely lattice-architecture module (sec. 2.2.1), and another is an examination of module symmetries to discover *module isomorphisms*. Both these exploitations help reduce the size of the searchable space the planner must navigate to discover reconfiguration solutions which ultimately decreases CSRP execution time.

4.2.1 Lattice Structure

Conforming a hybrid module to act purely within the its lattice framework over chain-based operations reduces planning complexity because:

1. For lattice architectures, configuration space is finite. Although Superbot is a hybrid module capable of lattice and chain type movements, this planner restricts module positions to discrete positions defined by the lattice grid, to make planning more manageable. This is a cubic lattice, of which each Superbot part occupies one cubic cell

2. Some planners like Million Module March (MMM) define the position of surface-moving modules according to which connecting faces they are connected to at any one time. However there can be instances of such modules having their connected

parts located in concavities (sec. 5.1.1); concave regions of the robot's structural surface where a single part of a surface moving module has the option of connecting to two or more connectors of structural modules. MMM, as a generalized-metamodule planner, treats the case of being connected to one connector or another as two separate states so as to not make any undue assumptions about module hardware. For Superbot however, it makes no difference during a reconfiguration which connector the connected part is connected to, only that it is the 'connect part'. In either case the mobile module is functionally identical, and is therefore considered to be in the same state, which is another way this planner uses symmetry to reduce the size of searchable state spaces[7]. An exception here is when a mobile module is in the presence of helper modules that moves also (sec. 4.4.1). If a mobile module is adjacent to a helper module which is about to move, there will be a different outcome if the mobile module was connected to it (it will be taken for a ride) or the static robot surface (it will remain where it is)

3. An additional benefit of the definition of a state's position raised in the previous point is there are also less connection-actions required for a planner to consider. Besides the inclusion of helper modules (discussed later) there is only one connection-type action the CSRP considers. This is simply to swap the connected part of a module such that the 'unconnected part' latched onto any structural modules around it, and the 'connected part' disconnects all its connection. If a state's position was defined by connectors instead of position, then the planner must know *which* connector the

[7]In fact this 'connected part' exploitation can be utilized by most module designs including ATRON, Fracta, MTRAN and Claytronics (sec. 2.3.3 - 2.3.4, 3.6). It does not however apply to Roombots (sec. 2.3.6) due to each part's degree of freedom that allows the connectors of a common part to rotate relative to each other (when that degree of freedom is actuated, the module's motion will be dependent on which connector was connected to the robotic structure)

previously unconnected part should latch onto instead of simply latching onto every connector it happens to be adjacent to. For a planner to decide this, the amount of connection-actions needs to increase, one for each connector of the unconnected part. This would increase the *dimensionality* of the state space. The *size* of the states space would also increase by argument of the previous point. So due to the *location*-definition of a module's position over the *connector*-definition, the CSRP's state space is smaller and less dimensional than otherwise

4.2.2 Module Isomorphisms

A Superbot module is non-isotropic, and so an SRP must take into account possible configurations this module can be in (bound by the lattice framework), as each is likely to be functionally different. This contributes to the problem of a large configuration space as discussed in sec. 3.5. However the configuration space can potentially be reduced by identifying geometric symmetries in a module. Certain symmetries are also *isomorphisms*; which cause every module-configuration to have a functionally isomorphic 'twin' (assuming the module cluster is homogeneous). Not every symmetry is a isomorphism, a combinations of symmetries is sometimes required for other isomorphisms discussed below. Since an SRP is concerned with module functionality only, it can collate each set of isomorphic configurations into a single *state* to consider as per the *state representation* of the module (sec. 4.3). Every isomorphism found will halve the state-space size compared to the original configuration-space. The Superbot module has three symmetries seen in Fig. 4.3;

1. Horizontal slice between both parts
2. Vertical slice of front face
3. Vertical slice of the side face

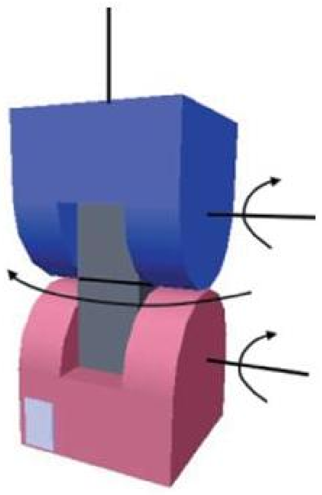


Figure 4.3: A module 3-R module with three degrees of freedom, from Fitch and Butler [14]

This first symmetry exposes the equivalence of the module parts, if the module was to be turned upside down, the respective part locations would obviously change but the module's state remains unchanged as the module's geometric and kinematic features are identical. This is one isomorphism. Another isomorphism can be found in each part using both vertical symmetries. If the module's central axle connecting both parts is actuated by 180 degrees, the revolved part will also be unchanged functionally. This applied for both parts, so the 3 isomorphisms founds in Superbot are:

1. Module parts are identical (the module can be turned upside down)
2. Top part can be spun 180 degrees
3. Bottom part can be spun 180 degrees

For every isomorphism identified, the state space size can be halved. So Superbot's three isomorphisms translate to an eight fold reduction of the searchable space of a homogeneous robot, greatly enhancing the CSRP's performance. Notice that the module's isomorphisms are distinct from its symmetries; the combined symmetries 2 & 3 were *both* required for isomorphisms 2 & 3 to exist; if the third symmetry is removed then neither isomorphism 2 or 3 can exist[8].

A final point is that even though an SRP can compute reconfiguration strategies using the state-space alone (without regard to the configuration space), when it comes to executing the physical reconfiguration the SRP must be able to determine the configurations of each module. Because even though a module part can spin 180 degrees and be in the same state, the polarity of actuators reverse, and by executing an action knowing only the module's state (not configuration), the actuator may turn the part in the opposite direction. Though this is easily fixed; the CSRP keeps a log of every module's present configuration, and the DSRP relies on every module keeping track of its own configuration. In this way action commands concerning a particular module states can always be translated into the action command concerning the module's exact configuration.

4.3 State Representation

As discussed in sec. 3.1.1, the *state representation* is the information needed to fully define a module's state; the functional (geometric and kinematic) features of its *configuration*. By exploiting configuration isomorphisms (sec. 4.2.2) the number of states a module can assume for a given position will be less than the number of configurations. To do this, the state representation should only include the minimum amount of information needed to identify a configuration's functional features. For example if the upper part in Fig. 4.3 is spun 180 degrees, the module is still in the same state but in a different configuration

[8]Proof: Using Fig. 4.3, if both parts' front faces developed equal bulges, then symmetries 1 and 2 would still hold, but not 3. In this case of rotating either part by 180 degrees about the central axis, the bulge would faces rearwards and the module would be geometrically altered. So isomorphisms 2 & 3 both fail

(the upper part is 180 degrees rotated relative to the bottom, not 0 degrees anymore). The information need to describe this change in configuration could be the facing direction of the front face, if it started off facing the ‘positive X’ direction, it will now be facing the ‘negative X’ direction. However to describe just the state, one only needs to record that ‘it faces in the X direction’ (the the positive/negative sign is superfluous information).

The Superbot module has been the module chosen for the CSRP’s implementation. It has 2 parts, 1 infinitely revolvable central axle common to both parts, and 2 axles inherent to each part which can rotate within a 180 degrees range as shown in Fig. 4.3. The planner defines the position of this module as the position of whichever part is connected to structural modules (surface moving module only ever need one part connected, as the other will be moving). The position of the connected part is defined by a universal Cartesian coordinate system, the origin of which is arbitrarily placed at one of the module-part locations at the commencement of simulation. A Superbot state is fully and uniquely defined by the following values:

- **Position (P):** universal Cartesian coordinate of connected part - *infinite possibilities*
- **Orientation Direction (OD):** which direction is the unconnected part relative to the connected part X, Y or Z? - *3 possibilities*
- **Orientation Sign (OS):** is the unconnected part more positive or negatively placed along the **OD** from the connected part - *2 possibilities*
- **Connected Part’s Axle Alignment (CPAA):** axle direction, can be X, Y, or Z direction as long as not the **OD** - *2 possibilities*
- **Unconnected Part’s Axle Alignment (UPAA):** same as above - *2 possibilities*
- **Connected Part’s Axle Rotation (CPAR):** -90, 0 or 90 degrees - *3 possibilities*
- **Unconnected Part’s Axle Rotation (UPAR):** same as above - *3 possibilities*

Using this coordinate system in a cubic lattice setting, a Superbot has 216 distinct states possible[9], and 1728 distinct configurations[10]. The reason why the CPAA or UPAA cannot be parallel to the module orientation direction (OD) is due to the Superbot hardware. Both part-axles are fixed perpendicular to the central axle, and the axle is always in the OD direction. Additionally, a *substate* is defined by elements of a state that concern just one of the parts. For example the unconnected part's substate would be made up of all the above except the CPAA and CPAR terms.

In the construction of an SRR's state network, the CSRP will first populate the network with legal states that a mobile module can exist in around the SRR's surface of structural modules. To do this the CSRP polls each connector of every surface structural module and computes the subset of possible states that can exists there given the presence of other modules close by. If the plane defined by the flat surface of the connector's face contains no other modules on the side opposite from the connector's part's location (this is true in either end module of Fig. 4.4a for example), then there are 108 possible states contributed. I.e there are 108 functionally unique ways a mobile module can attach itself to that connector[11].

[9]Proof: As seen in the Superbot state definition below, for a given module position, the combination of other state-defining possibilities is: $3 \times 2 \times 2 \times 2 \times 3 \times 3 = 216$

[10]Proof: a factor of eight greater than 216 states as found in sec. 4.2.2

[11]Proof: The module's OD can either be parallel or perpendicular to the normal vector $\vec{n}$ of the plane defined by the connector's flat surface. Case 1 ($OD||\vec{n}$); CPAA can be in one of two directions whilst attached (not three; it cannot be in direction $\vec{n}$). Case 2i ($OD\perp\vec{n}$,$CPAA\perp\vec{n}$); one connected-part substate exists here; the CPAR value that causes the connected part's face to face the connector (either -90deg or 90deg). Case 2ii ($OD\perp\vec{n}$,$CPAA||\vec{n}$); three connected-part substates exists here for all three values of CPAR. Also there are 4 combinations of OD and OS in the 'Case 2' type, i.e. a module can be perpendicular to $\vec{n}$ in 4 ways, and parallel 1 way. In all cases, the unconnected-part can be permutated through its independent terms; UPAA (2 possibilities) and UPAR (3 possibilities) totaling 6 independent possibilities. Thus $(1[2]+4[1+3])\times 6 = 108$

4.4 Motion Primitives

This section discussed some of the specialized motion primitive possible by a 3R module the CSRP is able to optionally utilize. A discussion of all fundamental motion primitives is in section 4.5.6.

4.4.1 Helper Modules

A helper module is one of the CSRP classifications of modules which helps mobile-modules by picking them up in one location on putting them down in another. The CSRP can be set to either include or exclude the helper-class.

Case 1: No Helpers

Planning with no helper modules is to plan for all mobilized modules to travel independently towards a goal region. This is a much simpler way to plan as no explicit coordination is required between modules. The downside of this method is the existence of many robot configurations that are impossible to reconfigure. For example, this straight line example of four Superbot modules seen in Fig. 4.4 is one impossible configuration. Here, the SRR is unable to fill the area marked with a wireframe with any of its modules. Due to global connectivity constraints; only the modules at either end of this line configuration are allowed to detach and become mobile in order to fill the wireframe. If either module in the middle tried this, the structure would be separated into two separate parts as shown in Fig. 4.4b and global connectivity is violated. However neither end module has to capacity to reach around to a new connector from where they are currently located, as seen in figures 4.5a - 4.5d, and thus are both stuck where they are. Hence the option of no designated helper modules can severely limit a robot's ability to reconfigure. The advantage of this case, if a

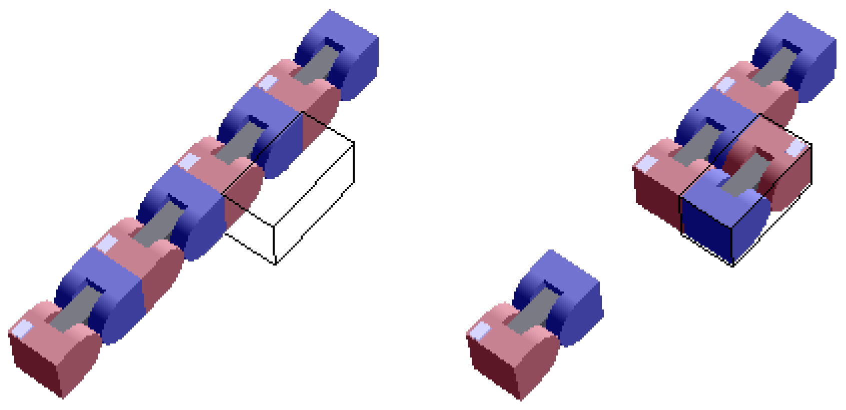


Figure 4.4: (a) A modular robot is in need of a *helper* module to fill wireframe box. (b) it cannot use a middle module as this would violate global connectivity

reconfigurations is still possible, is reconfigurations are much less computationally expensive to solve.

Case 2: Helpers Included

Reconfigurations from initial shapes such as Fig. 4.4a are possible with the inclusion of helper modules. This allows the end module to be able to ask the middle module it is connected to for help, by moving as well. In this case the middle module is acting as a helper module, and the end module is a mobile module. The helper module does not fully disconnect, one part has the ability to move and change connections (the *end* part (pink), which connects to the mobile module) but the other will remain static and fixed to the rest of the robotic structure (the *base* part (blue), always attached to the rest of the SRR structure). Fig. 4.5 shows a 15-atomic action process on how a light colored mobile module is able to travel to the wireframe with help. Figs. 4.5a - 4.5d show actions that do not yet require the helper's aid. This is the extent of the mobile module's reach, and so now requires the helper to move. Figs. 4.5e - 4.5g show how the extra reach provided by the helper module is able to align one of the mobile module's connecting faces with one of the helper's own connecting faces on its base-part, the part which remains static. Between Figs. 4.5g - 4.5h the mobile module first connects to the helper's base part, and then disconnects from its initial connection with the helper's end part, letting it return to its default configuration. From Fig. 4.5h onwards, the mobile module can move by itself, it does not require any more helpers. Travelling toward the wireframe in Figs. 4.5h - 4.5o requires an '*up and over*' type motion seen (similar to MTRAN's forward-roll motion; sec. 2.3.4). What Figs. 4.5j and 4.5n do not show is additional 'connection swaps', in which the front part (the part closest to the wireframe) has just moved into place and it connects to the structure, and then the rearward part can safely disconnect allowing it to swing 'up and over', bringing the module closer to its goal.

The example shown in Fig. 4.5 is the result of autonomous planning by CSRP. This required searching this module's state space, i.e. all possible states this mobile module can assume given that every other module is structural. If there are no helper modules, this space is relatively small, there are only 1512 states possible[12]. If two helper modules are considered by the planner however (either end module in Fig. 4.5o), the state space grows to 19380, a 12.8 fold increase in size. This is the disadvantage of including helpers in the search space, they augment the search task considerably. However they were necessary in this task to successfully fill the wireframe box with a module.

[12]Proof: As mentioned in sec. 4.3 there are 108 states for every connectable face assuming no other modules exist on the other side of the plane made by the connecting face. This holds true for every unconnected connector face in the 3-module line shape (the 4th module, the mobile module does not count, as this is the module of whom this state space is being constructed). All modules have 6 connectors but both end structural modules have one connector hidden from the mobile module as a static connection with the central structural module, so they only have 5 available connectors for the mobile module. The central structural module has only 4 connectors available for the same reason. Thus $108 \times (5 + 4 + 5) = 1512$

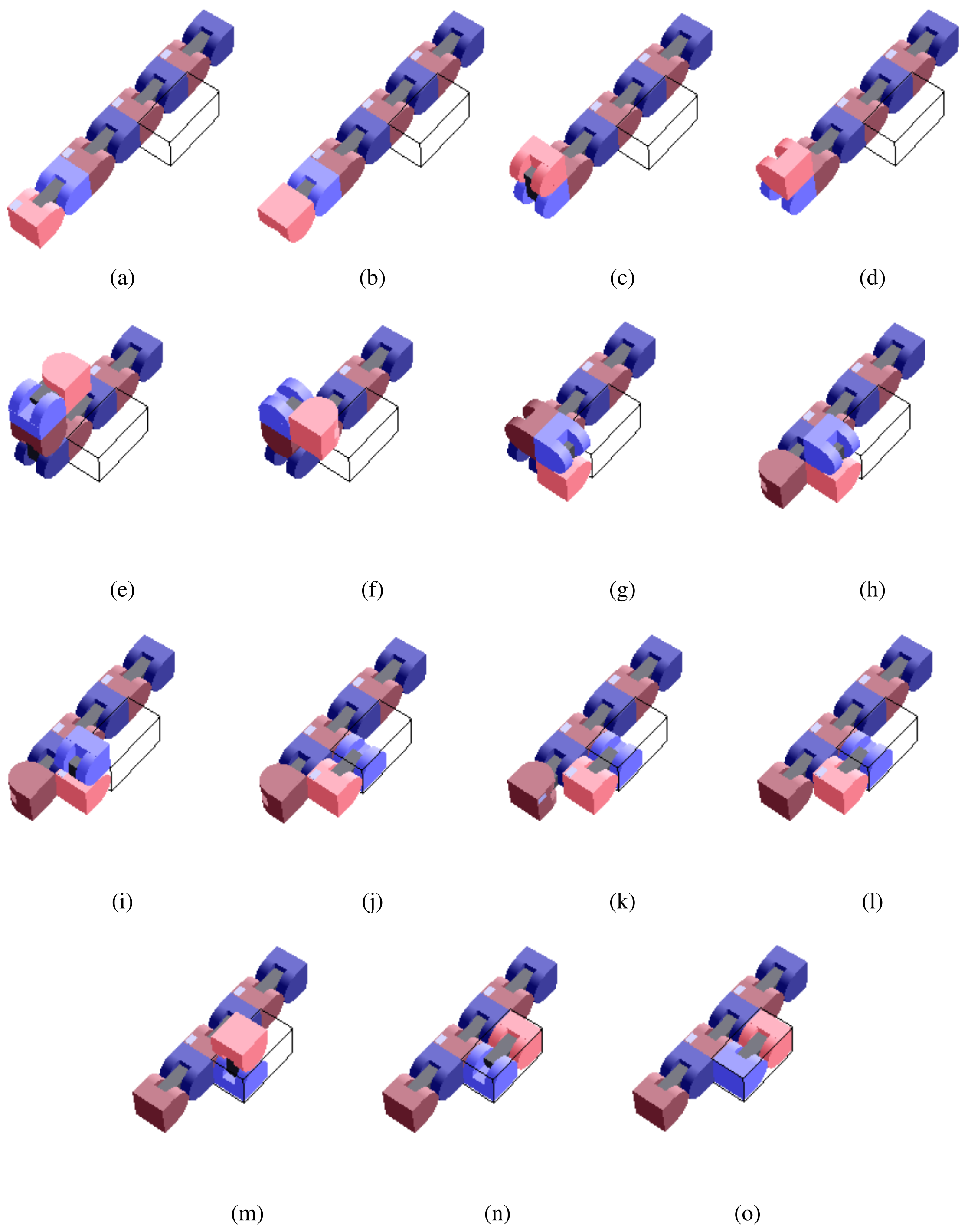


Figure 4.5: Helper module assisting a mobile module to relocate to the wireframe box

The reason the state space increases so dramatically is because the planner has to *coordinate* the actions taken by the mobile module and the helper module. As seen in Figs. 4.5c - 4.5d and then Figs. 4.5d - 4.5e, the mobile module was moving and then the helper module moved, and later the module moved again in Figs. 4.5h - 4.5i, which displays some of this coordinated behavior. In effect the planner is treating this as a 'super module', one that is made of two Superbot modules and thus has twice the degrees of freedom and twice the connectable faces. Doubling the degrees of freedom leads to an *exponential* increase in the number of unique configurations this 'super module' can assume, defined by an extension of the state representation of a single module (sec. 4.3). This is exponential because for every unique configuration one module is in, the whole set of unique configuration of the other module needs to be considered due to their interdependence of coordination. As discussed in sec. 4.3, there are 108 states a module can assume for every connecting face, thus for a mobile module connected to a helper's connecting face, and then the helper module connected to a static module's connecting face there are of order $\sim 108 \times 108 = 11664$ unique configurations for a helper and mobile module to be in[13].

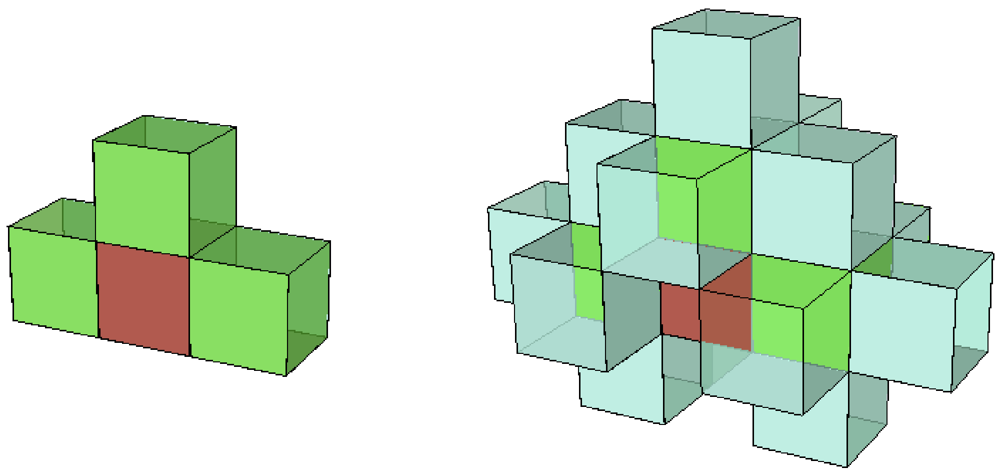
Implementation: Super States A 'super module' requires a representation for the planner to make sense of it, called a *superstate*. This requires both the Global Search and Local Search routines (sec. 4.5.3 & 4.5.5) to search a superstate space instead of the regular state space. A superstate always specifies what state a mobile module is in and sometimes specifies what state a helper module is in. If it does not specify what state a helper module is in, then the helper is considered to be in its *default* state; the state it was in initially when

[13]the exact answer is up to a maximum of 9510 unique configurations, depending how many non-mobile modules are around blocking configurations it could otherwise assume. Even with obstructing modules absent, 9510 is less than 11664 because a helper's base part is not allowed to move, so its side base part's connection faces are not contributing to possible configurations it could reach

the Global Search selected the mobile-module to mobilize. If it can be helped, a mobile module's superstate should not specify a helper's state, to limit the amount of super states a helper adds to the overall searchable space[14]. For instance, a mobile module far from a helper does not need to take into account the helper module's state as their actions are independent (their mutual distance prohibits any immediate collision events or chances of meaningful coordination). Thus the planner need not consider an extra $\sim 108 \times 108$ superstates for every connector 'far' away a mobile module can potentially connect to. The definition of 'far' here is defined by beyond the helper's *proximity*; the region around the helper where the actions of either module are mutually dependent, i.e. where they can possibly collide without correct coordinate of movements. A helper must consider all mobile module states that are in this region or one atomic motion away from being in this region. Fig. 4.6 shows this region of cells comprising a helper's proximity. By modeling mobile module & helper module representation like this, each helper only contributes 9510 super states max. to the state space.

A superstate is really a means in which a mobile module can control the helper module. A module is said to *engage* with a helper when it wants it to move, it can choose to do this when moving into the helper's proximity. This incurs a high computational cost of up to 9510 super states to consider, but is often necessary. Before a module engages with a helper it has 7 fixed actions it can take, in addition for 1 possible *engagement* action for every helper's proximity a mobile module is in (sec. 4.5.6). When connected to a helper, the module receives an additional 7 actions to control the helper and has one *disengagement* action to give up control of the helper it is engaged to. It was originally thought that a mobile module only requires control of a helper module when physically connected to its

[14]henceforth 'superstate space' will be referred to as 'state space' for brevity



(a) A helper module's possible part locations: the red cell marks helper's static base part, green cells mark the 3 possible location of a helper's end part can reach

(b) Helper's proximity: the additional blue cells mark the rest of a helper's proximity which includes all cells adjacent to possible end-part locations (green cells)

Figure 4.6: Helper module's proximity

movable end-part, as the example in Fig. 4.5 showed, this would be fine, as the mobile module it connected to the helper right up until it is dropped off onto another connector (Fig. 4.5g) from whence it can continue traveling without further aid. However if the task is reversed, and the module in the wireframe box of Fig. 4.5o attempted to relocate back to its original position in Fig. 4.5a it would be unable to do so. By progressing back through these figures 4.5o - 4.5a, it becomes apparent that the mobile module needs to be able to command the helper module to first reconfigure before it can connect to its end part Fig. 4.5g before it physically connects to the end part. So the proximity notion was used which allows the task in Fig. 4.5 to be executed forwards or backwards without a problem. A final note; as the superstate is really just a state with more information, the word 'state' is often used in place of 'superstate' in the rest of this report for brevity.

4.4.2 Simultaneous Actuation

Degrees of freedom help define atomic movements a module it can take, such as the bi-partite MTRAN (sec. 2.3.4) which can actuate any two of its axles by 90 degrees in either direction providing it with max. four choices of atomic motions in any one state. However simultaneous actuation can provide a module with more reconfigurable options than corresponding succession of actuations allow. For example, figure 4.7 shows module AB attempting to move away from module CD. From an initial configuration in Fig. 4.7a, if part A were to turn in either way (clockwise/anticlockwise) whilst part B remained inactive, part B would collide with part C. Similarly if part B attempted to turn either way whilst part A was inactive, then parts B and C would also collide. Therefore all four atomic actions of MTRAN module AB result in collisions, and module AB is unable to disconnect and move apart from module CD. Simultaneous Actuation can help here. As seen in Fig. 4.7b, if parts A & B turn anticlockwise (ACW) & clockwise (CW) respectively, part B can safely move away from part C because their facing surfaces remain parallel. Because this double-action results in behavior inaccessible from successive actions, it must be classed as a distinct atomic action itself. Hence if simultaneous actuation is enabled by a planner, there is 4 additional actions[15] to consider at any one module state making 8 in total. In fact the CW/CW and ACW/ACW simultaneous action pairs never useful and can always be replicated by successive single actions, so only 6 defined actions total are needed to account for simultaneous actuation of MTRAN modules.

Simultaneous actuations in the CSRP is optional, selected by the user before program execution. The advantage of simultaneous actuations is a robot has more reconfigurable

[15]which are two parts turning CW/CW, CW/ACW, ACW/CW and ACW/ACW

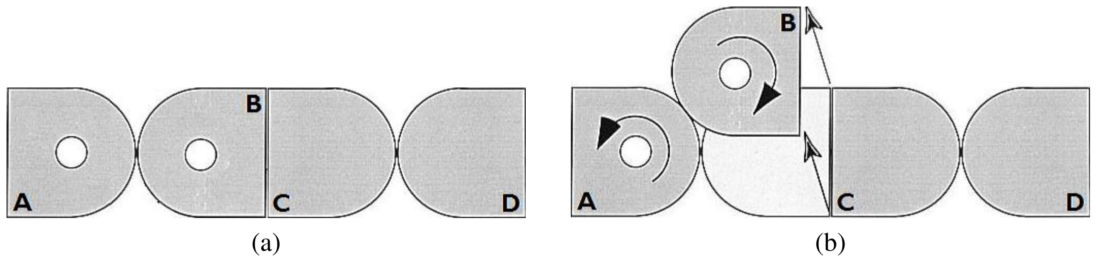


Figure 4.7: Simultaneously actuated motion of a MTRAN module to avoid a collision, (edited) from Murata *et al.* [28]

options. Thus a planner can sometimes find shorter reconfigurations paths (less atomic actions required to reconfigure) and a small fraction of reconfiguration tasks become possible that would not be possible otherwise. The cost of these benefits is the increase of the dimensionality of the searchable space, increasing execution time. Simultaneous actuation does not increase the amount of configuration nodes in the configuration space (Fig. 1.4) but does increase the amount of links between nodes of which the planner must examine at each state node. This may seem a fair price though, because it is not necessarily a good idea to be restricting non-simultaneous actions just for the sake of reducing the dimensionality of the searchable space so a planner can plan faster slightly faster. The entire reason for a planner is to exploit knowledge of the hardware such that less moves are required to reconfigure a robot saving on resources that really matter to a robot like power and time (the time to execute a reconfiguration in hardware is often orders of magnitude greater than time needed to discover the reconfiguration solution in software, so often more time invested on computing a better reconfiguration solution is time saved overall). The downside of simultaneous actuation is that the reward/computational-cost ratio isn't very high. For the far majority of reconfigurations, at least for the Superbot module tested in this thesis simulation, the planner does not find shorter solutions using simultaneous actuation. Their inclusion often ends in wasted computational effort, increasing execution time by 29%.

4.5 Algorithm

The CSRP, written in Java, comprises several important functions which this section serves to break down and explain. This is in roughly chronological order in which each function is called, which is top-down. The CSRP begins with input from a user into a GUI (Graphical User Interface) specifying a shape-goal for the robot to morph into, then conducts a hierarchical search in native kinematic space to find a reconfiguration solution. The source code of all routines discussed in this section are also available in the attached DVD.

Program flow is shown by Fig. 4.8. The preamble of the program involves waiting for a user to specify a *final shape* in the GUI, which Tile Pattern converts into a *final configuration* that jig-saws the modules into the final shape. At this point the hierarchical search begins. First, Global Search mobilizes modules that are allowed to detach and gives them goals location to move to as part of the reconfiguration process. To check if modules are detachable, Global Search uses the function *Connectivity Checker*. Global Search passes each module relocation task to Local Search which searches through the module's state space to find an optimal transition path for the module to get to its given goal location. For Local Search to search the state space of a module, it requires a state representations and knowledge of the module's native kinematics which the Transition Model provides. Transition Model bundles all three hardware-dependent routines; *Transition*, *Collision Detection* and *Translation* together with some other smaller routines (such as those which ensure a module cannot take actions that exceed their hardware-defined restrictions). Transition determines what states link to what states in the searchable state network. Collision Detection ensure these links (module actions) are collision-safe, otherwise they are pruned in the search. Translation serves to take a state-defined action, and translate into a more information rich configuration-defined actions for reasons introduced in sec 4.2.2. Each routine is described in greater detail below, beginning with a single line function description and accompanied with pseudo-code where appropriate.

Currently the three routines of Transition Model are partially hard-coded specific to the Superbot Module. In order to achieve the thesis' ultimate goal (sec. 1.2) of generality whilst planning in native kinematic space, these functions will have to be fully automated. This means they must be able to operate purely from a representation of a module's design without relying on any hard-code. This is left as future work (sec.6.3.3).

4.5.1 GUI

GUI (userInput) $\longrightarrow$ (Positions) finalShape

The Graphical User Interface (GUI) provides an easy means of submitting a shape goal into the CSRP. The details of its implementation are not relevant to the planning aspect of this thesis, but this brief section serves to give an idea of how reconfiguration simulations are initiated.

Fig. 4.9 shows the simulation initiating with a line of Superbot modules (Fig. 4.9b) and the corresponding GUI (Fig. 4.9a) where the cells the robot is currently occupying are grayed in. As the robot was chosen to start with 4 modules, it has 8 parts in which to make an 8-cell shape. This is selected as the cyan cells in Fig. 4.10a. This is the only information the planner receives from the user. These cell positions are saved into a collection called *finalShape* which is passed to the Tile Pattern routine (sec. 4.5.2), which works out how modules can be jig-sawed together to form such a shape, and thus derive a *configuration goal*. Once Tile Pattern has been called, and the rest of the CSRP executes and the SRR reconfigures into the desired shape (Fig. 4.10b).

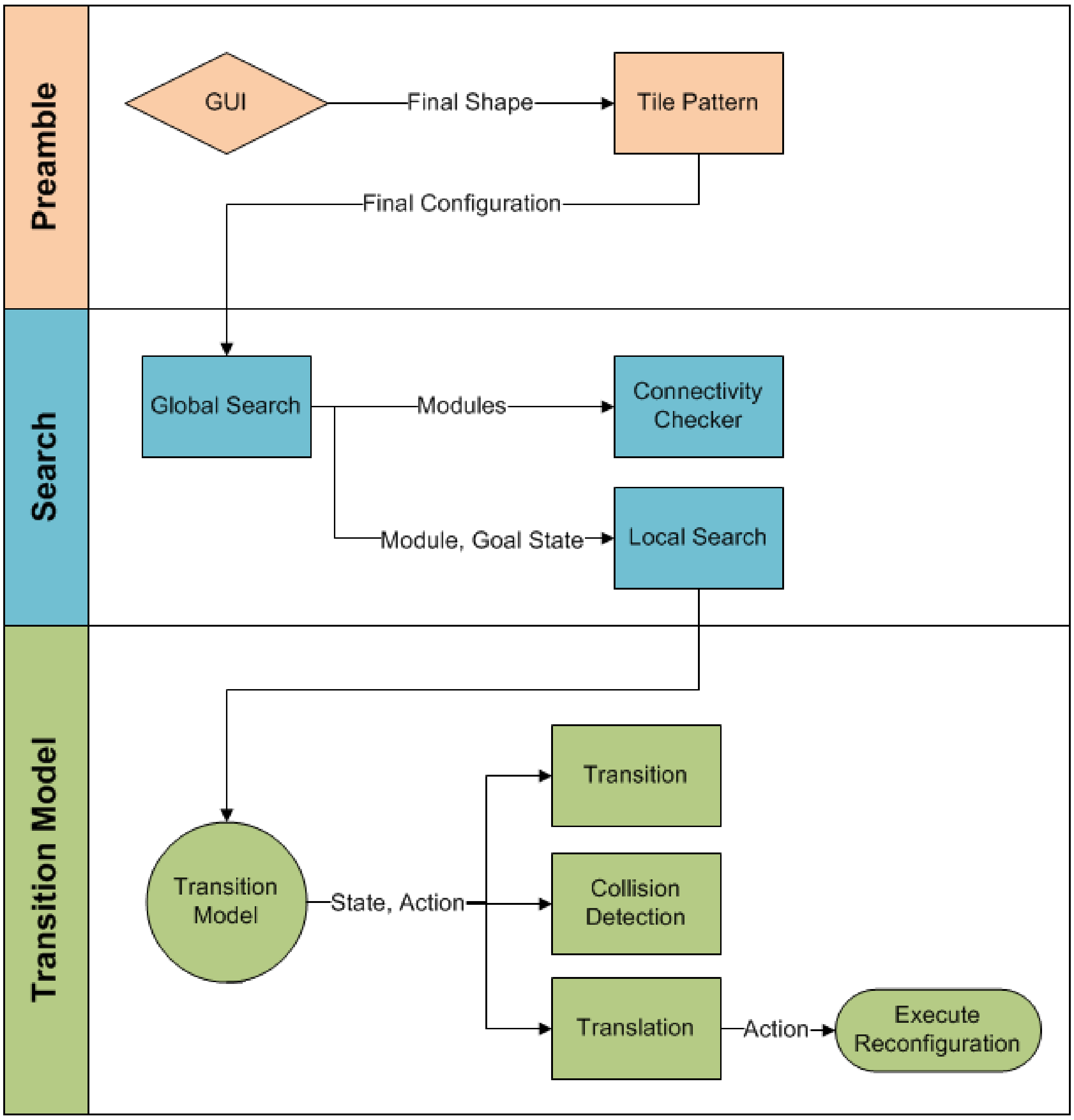
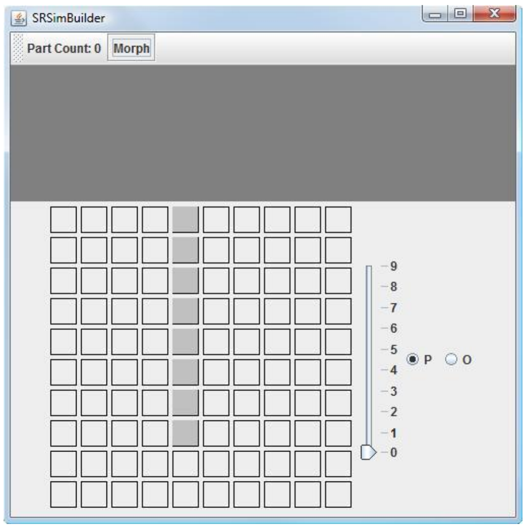
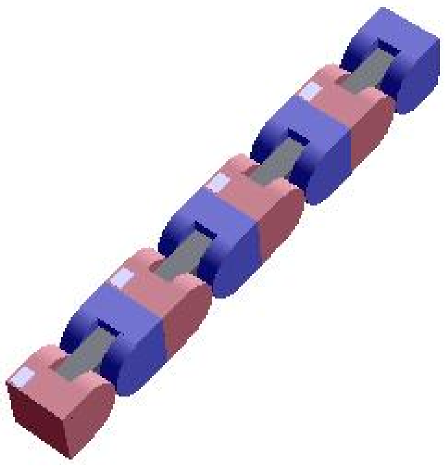


Figure 4.8: Program Flow: Organization of CSRP routines

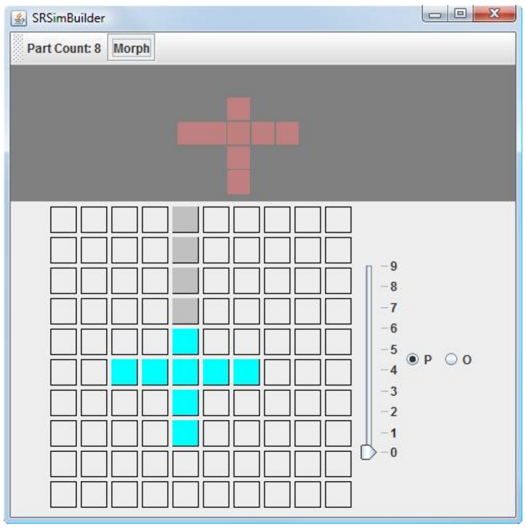


(a) GUI

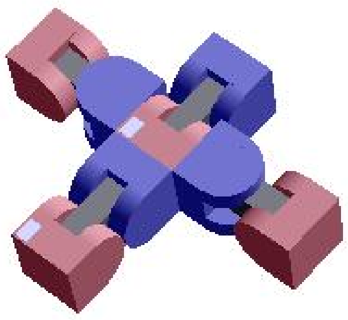


(b) Robot

Figure 4.9: GUI and corresponding modular robot: Initial state



(a) GUI



(b) Robot

Figure 4.10: GUI and corresponding modular robot: Reconfigured state

4.5.2 Tile Pattern

tilePattern (finalShape, directionalPreference) $\longrightarrow$ (Tiles) finalConfiguration

Configuration goals are ideal for a reconfiguration planner, as they are exact specifications of goal-states the planner should search for amongst the searchable network of state-nodes. Unfortunately a planner cannot expect a higher level program installed in the modular robot to pass it a *configuration* argument to morph into. This is because configurations are hardware-knowledgable representations and part of an SRP's role is to abstract away all details to do with hardware specifics for any higher level program calling it. So this thesis planner is designed to accept a *shape goal* instead from which it then attempts to determine how to piece rectangular bipartite modules together to fit into and assume the given goal shape. *Tile Pattern* is a routine written to handle this; it finds a configuration-goal given the 3D shape-goal inputted into the GUI.

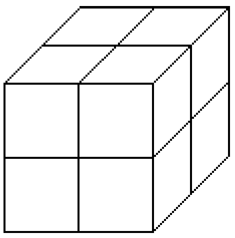
If all final configurations that fill such a shape are equally desirable, then Tile Pattern must find at least one configuration from a set of valid robot configurations that fill this shape. Although if one wishes to deal with the fact that different configurations will require a different number of modular movements to transform into, the problem gets more complex. As always; the least amount of module movements would be ideal, saving time and power, but which configuration solution this is, is not entirely obvious. This is one challenge a shape-goal poses to Tile Pattern, a precursor to this is determining the set of configurations that fit inside the shape goal in the first place.

For bipartite modules, this problem of how to jig-saw modules together to fit inside a lattice-defined shape is closely related to a field of mathematics called *domino tiling*.

Domino tiling examines the numbers of ways a particular shape in a square or cubic lattice can be filled with rectangular dominos that are 1 cell in width and 2 cells in length, just like MTRAN, Superbot and Roombot (sec. 2.3.4 - 2.3.6). Unfortunately the set of solutions found by domino tiling explodes with an increasing shape size. For example; a $2 \times 2 \times 2$ cell shape as shown in Fig. 4.11a has 9 unique tiling solutions. Doubling the dimensions and a $4 \times 4 \times 4$ shape has 5.05 billion ways it can be tiled, of which Fig. 4.11b shows one example. Extending two of those dimensions to a $6 \times 6 \times 4$ shape there are 1.23×10^{23} valid tiling solutions [13, p.408] [35, p.759]. The challenge this poses is that the set of configurations (tiling solutions) for large modular robots is too many to consider comprehensively. Several configuration solutions can be considered to select from (to choose whichever offers the shortest reconfiguration path[16]), but definitely not all. Thus a compromise is needed to balance the number of goal-configurations worth checking against the computational cost of doing so. There will exist some number of configurations that is 'reasonable' to check before the best reconfiguration solution found should be executed as the probability of finding increasingly more desirable goal configurations decreases with each examined. What defines 'reasonable' above is still an open problem in this thesis and is important one as the planner's overall performance is very much dependent on it.

Currently Tile Pattern has only been implemented to the stage where it will find up to four configuration goals when passed a shape goal to tile. The different configurations goals it will return is dependent on an optional *direction of preference* argument which is: if the routine comes across multiple ways in which to tile a certain region of the shape, is will align the rectangular modules in this preferred direction. This routine was created by

[16]shortest reconfiguration path is the minimum distance between two state nodes in the robot's state network which translates to the least numbers of atomic actions required of the robot to transform from one configuration to another


(a) A $2 \times 2 \times 2$ cell cube. There are 9 ways to tile this shape

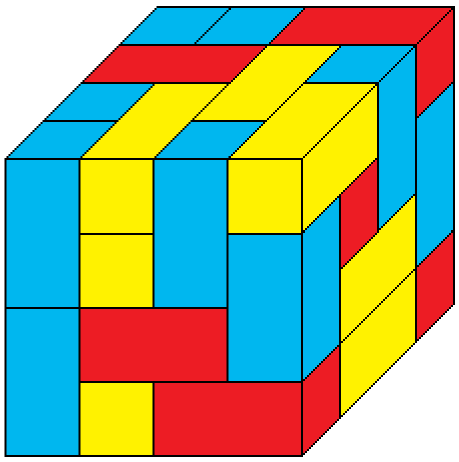

(b) A $4 \times 4 \times 4$ cell cube. One of 5,051,532,105 ways to tile this shape

Figure 4.11: Domino Tiling

the author, it is a *correct* algorithm albeit not a *complete* one, but nevertheless has shown to be quite robust in finding tiling solutions to the vast majority of shapes which are possible to tile.

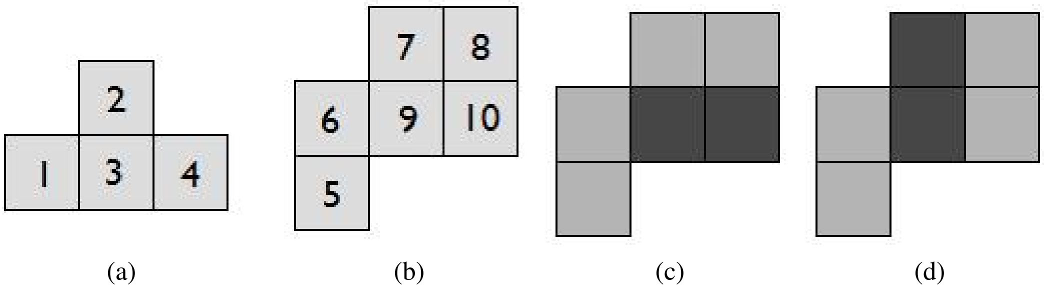


Figure 4.12: Tile Pattern case examples; (a) an impossible shape, (b) a possible shape, (c) tiling with horizontal tile preference, (d) tiling with vertical tile preference

Algorithm 4.1: ***Tile Pattern***

Tile Pattern begins by searching out over a particular shape, such as those shown in Fig. 4.12, and selects couples of cells to *tile* which becomes a reserved region in 3D space for a bipartite module to be placed. Fig.4.12a is an example of an impossible shape to tile; if one tiles adjacent cells 1 & 3, then 2 & 4 cannot tile, they are not adjacent so a bipartite module count not exist between them. Other trials are fruitless as well, tiling cells 2 & 3 is fine but then cells 1 & 4 are left separated. Fig. 4.12b is a possible shape to tile, in fact there are two ways to solve it as shown by Fig. 4.12c & Fig. 4.12d, however the shape must be tiled in a particular order to converge on a solution. If the routine first tiles cells 6 & 9, then it will ultimate fail as it has isolated cell 5. To prevent this, Tile Pattern prioritizes cells with the least neighbors (cell 5) to tile first (line 3). In order not to create an impossible shape later on (as cells are 'removed' from the consideration once tiled) the cells next selected will tile up with cell-neighbors that have the least amount of neighbors themselves (lines 10-14). In Fig. 4.12b, cell 5 does not have a choice, it only has cell 6 to tile with. Now there is a choice, the search is left with 4 cells (cells 7-10) to tile, all have an equal amount of neighbors. In this case one is randomly selected (whichever comes off the Queue first: line 6), say cell 7, and given that both its neighbors (cell 8 & 9) have two neighbors each, both cell 8 & 9 become equally weighted options to tile with. In this case, a *direction of preference* (line 12) specifies which neighboring cells to tile with, which is the planner's preferred direction to align modules in. Fig. 4.12c shows the result of a horizontal direction of preference, Fig. 4.12d shows vertical preference.

Algorithm 4.1 Tile Pattern

```
1: for each Part in finalShape do
2:     build list of neighbors
3:     place Part into Queue that prioritizes by least number of neighbors
4: end for
5: while Queue is not empty do
6:     pop next Part off the Queue
7:     if Part has no neighbors then
8:         return 'FAIL: finalShape is impossible to tile'
9:     end if
10:    find which of Part's Neighbors has the least neighbors itself
11:    if there is a tie then
12:        select the Neighbor that lies along Part's DirectionOfPreference
13:    end if
14:    tile(Part, Neighbor)
15: end while
```

4.5.3 Global Search

globalSearch (currentConfiguration, finalConfiguration) $\longrightarrow$ **void**

Global Search encompasses most of the SRP's work. It requires the *GUI* and *Tile Pattern* routines to have specified the final configuration of the robot, from whence it begins searching the robot's configuration space for a close-to-optimal path between the robot's current configuration and the (desired) final configuration. This function is part of a two-level search hierarchy. Global Search continually searches for modules that can safely detach (without causing global disconnection of the robot) and *sub goals* they can relocate to to realize a reconfiguration. To relocate modules, it calls upon *Local Search* to find valid transition paths between modules' current and (given) goal states. *Local Search* is called multiple times by Global Search, and only by Global Search, and so is the subordinate routine in the CSRP's search hierarchy.

In terms of the searchable state space, Global Search dynamically places subgoals within this space for the Local Search to link up by using regular search routines such as breadth first search (BFS). If Local Search is unable to find such a subgoal, Global Search has some capacity in which to dynamically reposition another subgoal and Local Search can have another try. What this is, is Global Search telling Local Search to find a particular path from a certain module to a certain location, if Local Search is unable to do this (because it is kinematically impossible to do so) then Global Search will select another module to fill that same location, and execute Local Search again (algor.4.2[line 20]).

Algorithm 4.2: ***Global Search***

Using the final configuration goal (line 2), Global search will continuously place subgoals in the searchable state space until a reconfiguration is complete (line 3). Each iteration of Global Search first identifies a set of modules that are allowed to detach and become mobile (lines 4-8). These modules have to satisfy the *Connectivity Checker* routine, but additionally not yet be a part of the goal configuration (the CSRP rejects these on the ground of planning backwards[17]). Using one of these mobile modules, the next (*tile*) of the final configuration that is not yet filled with a module can then be filled by one of these potentially mobile modules (lines 9-13). Only the tiles that are physically reachable by mobile modules are considered (line 10), tiles further out from the SRR structure wait until interim tile locations fill with modules until they also become reachable. This is the point where the CSRP differs much from a DSRP because now Global Search attempts to pick one potential module and one tile to match. The DSRP mobilizes multiple modules and directs them to move towards the multiple tiles simultaneously.

However for the CSRP, a *goal tile* is picked, using an order defined by the coordinate

[17]modules already in the goal state do not necessarily have to be rejected, their temporary use could be beneficial to other mobile modules however this is a complex matter discussed more in sec. 4.6.1

system (line 14). This ensures the packing process of a shape with modules happens uniformly and there are no pockets or cavities left in the structure that later become impossible for modules to move into. Once the *goal tile* is chosen, the closest potential module will attempt to fill it using Local Search (lines 18-23). If this Local Search is unsuccessful then the second closet mobile module will try, and so on. If however the module selected is the foundation in which it needs to reach the goal tile it will be discounted, because one module cannot support itself (be in 2 places at once). So this particular module is removed from the list of potential mobile modules due to goal-tile placement (lines 15-17). If one of these mobile modules is able to fill the *goal tile* from the final configuration, Local Search returns a positive flag (line 20) and so Global Search begins a new iteration (line 3) to search for the next potentially mobile modules and suitable tiles. Else, Global Search fails and displays that it is unable to find a valid reconfiguration solution (line 25).

4.5.4 Connectivity Checker

connectivityChecker (Module, CurrentConfiguration) $\longrightarrow$ (Boolean) IsDetachable

Identifying Surface Moving Modules

Routines for ensuring global connectivity have been borrowed from the *Million Module March* (MMM) planner (sec. 3.7) and adapted from cubes to modules. This routine attempts to finds *connection cycle* (sec. 3.7.2) among a module's neighbors, which is found, guarantees a module can safely become mobile and move away whilst conserving global connectivity of the modular robot. Hence this routine is called to identify potential surface moving modules to begin a reconfiguration.

Algorithm 4.2 Global Search

```
1: FinalShape = *GUI*(userInput)
2: FinalConfiguration = *tilePattern*(FinalShape,DirectionalPreference)
3: while CurrentConfiguration $\neq$ FinalConfiguration do
4:    for each Module in CurrentConfiguration do
5:       if (*connectivityChecker*(Module, CurrentConfiguration) = isDetachable AND
         Module is not yet a tile in FinalConfiguration) then
6:          add Module to set of potential mobile modules
7:       end if
8:    end for
9:    for each Tile in FinalConfiguration do
10:      if Tile does not contain a module AND Tile is adjacent to a current Module then
11:         add Tile to set of potential sub goals
12:      end if
13:   end for
14:   GoalState = tile from set of potential sub goals with highest X-coordinate, then Y-
      coordinate, then Z-coordinate
15:   if (any Module is the only way a module in GoalState would be connected to Robot
      AND Module is a member of set of potential mobile modules) then
16:      remove Module from set of potential mobile modules
17:   end if
18:   place each potential mobile module into Queue ordered closest to farthest distance
      from GoalState
19:   while Queue is not empty do
20:      if *localSearch*(next Module in Queue, GoalState, CurrentConfiguration) =
         (Boolean) successful then
21:         BREAK
22:      end if
23:   end while
24:   if no local searches successful then
25:      return FAIL: '*No module was able to reach the last goal state*'
26:   else if CurrentConfiguration = FinalConfiguration then
27:      print SUCCESS '*Robot reconfiguration complete*'
28:   end if
29: end while
```

Identifying Helper Modules

Once surface moving module(s) have been identified and given permission to mobilize, the next step is to identify helper modules (sec. 4.4.1) to help their motion. Identifying helpers is a similar process, requiring only one part of the helper (the *end-effector*) to be able to disconnect from a structure and move around. This is a more lenient connectivity test, and all potential surface moving modules have are also potential helpers, but not all helpers are able to be surface moving modules. Fig. 4.13 shows an example of a potential surface moving module acting as a helper; as such either part can be the end-effector or base. To make sense of this, the planner considers these as entirely different helper modules. Both modules on the end of the module chain (top-right or bottom-left) are also potential helpers, though only their unconnected parts can be end-effectors else they would disconnect themselves.

Algorithm 4.3: *Connectivity Checker*

Connectivity Checker begins searching out from each connected neighbors to the module in question, to see if they are independently connected to each other, either directly or indirectly through the presence of other modules. A unique search (Breadth First Search (BFS), line 3) begins at every neighbor, referenced by the neighbor module's unique ID number. Searches keep tabs of which modules they visit on a list (lines 4,17), which is visible to all searches. If one search finds another as it spreads out through the modular cluster (line 14), they *join*; in which one adopts the reference ID of the other (line 15). If all BFS searches from each neighbor eventually amalgamate into one, this guarantees the presence of a connection cycle (sec. 3.7.2) and the module in question can become mobile without violating global connectivity.

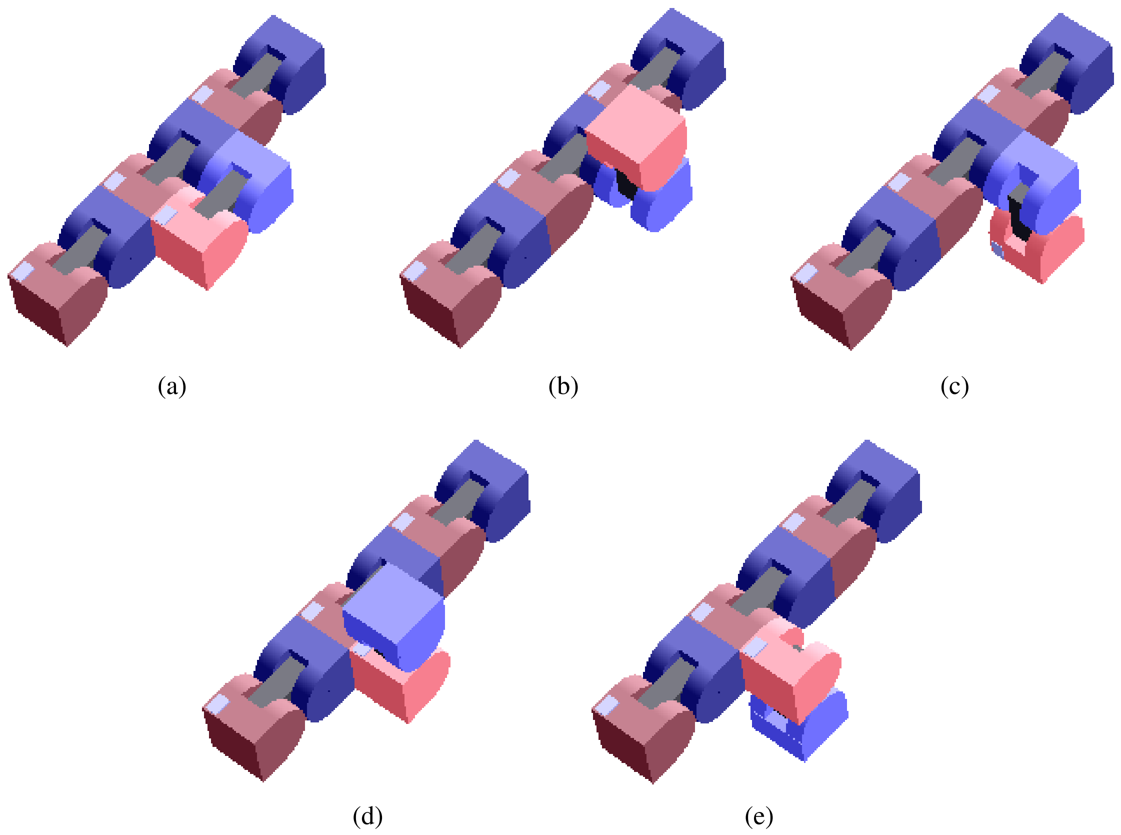


Figure 4.13: One module acting as two helper modules, (a) default position, (b) & (c) show possible configurations of one helper module, (d) & (e) show possible configurations of the second helper

Algorithm 4.3 Connectivity Checker

```
1: initialize list of visited modules
2: for each Neighbor of Module do
3:    initialize a BFS[Neighbor]
4:    visit(Neighbor, 'Neighbor ID')
5: end for
6: while depth < maximum search depth do
7:    for each Neighbor of Module do
8:       Parent = next module from BFS[Neighbor]
9:       Children = getNeighbors(Parent, CurrentConfiguration)
10:      for each Child do
11:         if Child = Module then
12:            continue
13:         end if
14:         if Child has been visited by a Different Neighbor then
15:            join(BFS[Neighbor], BFS[Different Neighbor])
16:         else
17:            visit(Child, 'Neighbor ID')
18:            add Child to BFS[Neighbor]
19:         end if
20:      end for
21:   end for
22: end while
23: if all BFS[Neighbors] have been joined into a single BFS then
24:    return true: 'Neighbors found each other independently of Module'
25: else
26:    return false: 'Module is required for global connectivity'
27: end if
```

4.5.5 Local Search

localSearch (MobileModule, GoalState, CurrentConfiguration) $\longrightarrow$ (Boolean) isSuccessful

Local Search is used to relocate modules from their current states to goal states, given by the Global Search routine. Its search is through native kinematic space, building a state network[18] respective to one mobile module outwards from the goal state over the structural topology (all other modules) of the SRR. When Local Search comes across the module's current state (such as that exampled by Fig. 4.2) it stops. Local Search uses aspects of dynamic programming (sec. 3.2.3) from this point, as Million Module March (sec. 3.7.2) also does, to propagate out a navigation function from the goal state. This allows any mobilized module to navigate towards the goal states by considering the *value of desirability* that the navigation function has labeled each state with (stored in a hashtable). This allows mobile modules to make optimal decisions in regards to locomotion, in order to relocate to their given goal state in a minimum number of atomic actions. Local Search is both a *correct* and *complete* algorithm in this regard.

The navigation function in Local Search begins with the goal state (passed in from Global Search), and assigns the goal state with value '0'. To calculate the values of other states; every state-transition incurs a penalty cost of '-1' to represent time and energy for a module to physically transition to another state. As this navigation function propagates outwards from the goal-state, its immediate state-neighbors will have values of '-1', and *their* neighbors will have value '-2'. An exception here is if that neighbor's neighbors is the goal state again. To get around this problem, states will only try opt to increase their current

[18]as discussed before, a state network is a collection of states a mobile module can be in. The links between state-nodes are valid actions a module can take from a respective state-node

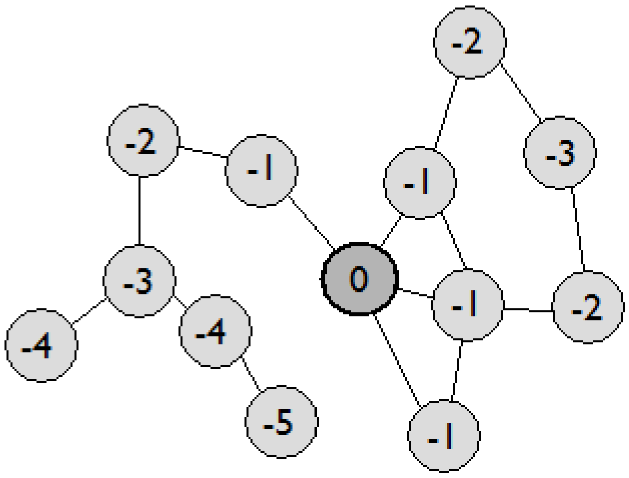


Figure 4.14: State network example: desirability values mark each state for a mobile module to consider (states are represented by circles, and module actions are represented by lines which link states)

value function. I.e. the goal state will not assign itself a value of '-2' if it is already a '0'. Likewise if a state in that state space has not been searched before, and has two neighbors that are valued '-5' and '-9', it will be labeled a '-6', due to selecting the best neighbors and adding on a transitional cost of '-1'. An example diagram of such is Fig. 4.14, the darker shaded circle-node representing the goal state.

Algorithm 4.4: ***Local Search - Find Path***

Local Search being searching from the goal state outwards to the current mobile-module state. Hence goal is the first entry added into a queue of state nodes to expand (line 2). The search will continue searching whilst there are still state nodes reachable to a hypothetical module at the goal state (line 3). For a modular robot of a few 10's of modules, there is no more than several 100,000 states and so the search will still commence within a short amount of time if it cannot find the current mobile module state (line 20). The search progresses by expanding the nodes of every state pulled off the front of the queue (lines

4-7). *Transition Model* is used to determine if a particular action from a state is valid given the current robots configuration (line 6). This checks:

1. The mobile module is not overextending its degrees of freedom if they have restrictions (inherent to a module's design)
2. Taking such an action will not collide with anything by using the *Collision Detection* routine discussed in sec. 4.5.7

A double link is created between each expanded node and its parent (line 8), such that executing of the search path afterwards is quick. If the search reaches the current module state (lines 9-12) it will call on algor.4.5 to execute the module's motion. Expanded child states are then given a value of desirability (lines 13-16). This is the dynamic planning in action, which would create a state network resembling something like Fig. 4.14. The search will continue expanding nodes until it exhausts them all, and which case it fails (line 20).

Algorithm 4.5: ***Local Search - Execute Path***

Executing the Local Search is a matter backtracking through the state *values of desirability*. This is quick routine linearly proportional to the number of states the module will travel. While the mobilized module is not yet at the goal state, it will search out each state-neighbor it can transition to in one atomic action (lines 2-4). It will then attempt to execute the action, but because this module has taken advantage of isomorphisms such *state-action* needs to be translated into a *configuration-action* (line 5). A description of the Translation routine (sec. 4.5.8) discusses this further.

Algorithm 4.4 Local Search: Find Path

```
1:  build list of visited states
2:  add GoalState to Queue
3:  while Queue is not empty do
4:     State = pop off next Queue element
5:     for every Action a module can take do
6:        if transitionModel(State, Action) = (Boolean) valid action then
7:           ChildState = transition(State, Action)
8:           add ChildState to State's list of neighbors and vice versa
9:           if ChildState = the current state of MobileModule then
10:             CALL executePath(MobileModule, GoalState)
11:             return SUCCESS: 'Local Search linked MobileModule to GoalState'
12:          end if
13:          if (ChildState has not yet been visited OR value of ChildState $<$ value of State
             - 1) then
14:             value of ChildState = value of State - 1
15:             add ChildState to back of Queue
16:          end if
17:       end if
18:    end for
19: end while
20: return FAIL: 'exhausted all search paths, no transition path was found'
```

Algorithm 4.5 Local Search: Execute Path

```
1: while MobileModule state $\neq$ GoalState do
2:    for each neighboring state of the state MobileModule is currently in do
3:       find the most desirable neighbor state to transfer to (the most value)
4:    end for
5:    EXECUTE translation(Action which transfers MobileModule's current state to most
      desirable neighbor state)
6: end while
```

4.5.6 Transition

transition (state, action) $\longrightarrow$ (State) newState

Transition answers the following; if a module were in a certain state, and it took a certain action, what new state would it end up in? The planner's Local Search needs this function when expanding state-nodes as part of its search routine. Expanding a node means to look through all valid actions that state can take, and to know which state-nodes they lead to. Transition is hardware-dependent, its implementation is ultimately dependent on how a module's state is defined (sec. 4.3). It is also dependent on the native kinematics of the module, i.e. what actions are available to the module in such a state. Usually taking different actions will lead to different module-states, otherwise they would be redundant and increase search complexity unnecessarily[19]. To plan reconfiguration of robots of arbitrary module designs, *Transition* needs to be automated. This is a complex task, and discussed further in sec.6.3.3.

There are 16 fixed state-actions this thesis planner allows a Superbot module to take as indexed in table 4.1. Additional state-actions also become available for every helper's proximity[20] a mobile module is in. These are for the ability to *engage* with a helper for aid in relocating as discussed in sec. 4.4.1. Actions -2, 0 - 5 & 7 (table 4.1) are always available to a module to execute as long as *Transition Model* verifies they are safe/possible. Some action are not possible due to a module's degree of freedom restrictions. Both Superbot part's are able to move about their part-axles over a range -90 degrees to +90 degrees. Hence if the mobile module's unconnected part is at +90 rotation about its part axle already

[19]in this thesis planner's implementation with Superbot modules, sometimes actions 4 & 5 as listed below lead to the same state, but not always, it depends on how both parts are oriented about their degrees of freedom. It also depends on whether anything is connected to the module, such as if it acting as a helper module. For these reasons both actions 4 & 5 are kept

[20]defined in sec. 4.4.1

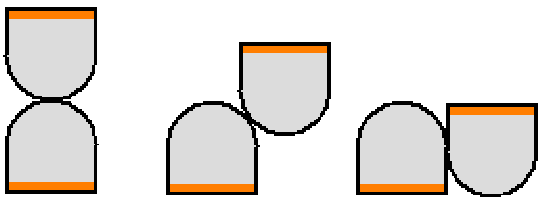


Figure 4.15: An example of Superbot performing a simultaneous move, in which the unconnected part always has its face directed in the same direction

say, a simple check by *Transition Model* would inform Local Search that 'action 0' is not valid, as this would attempt to turn the part forwards[21] beyond +90 degrees, which the hardware will not allow.

Actions in 8 to 15 become available to a mobile module when engaged to a helper, this allows it to control the helper's state in the same way it controls itself, as seen in the resemblance of action descriptions 10-15 and 0-5. When engaged, a module also has the ability to *disengage* from its helper (action #16), in which it releases control of the helper. This is enforced by the planner when a mobile module attempts to move outside the influence of a currently engaged helper module to reduce the size of the searchable state space (discussed sec4.4.1). When a module is not engaged to any module, it may have several engaging actions. For example if it is in the proximity of four helpers, actions 16 - 19 become available, each of which will engage the mobile module to a respective helper module. Action 6 is a simultaneous action was available to mobile module but not helper modules yet. Their inclusion is optional and is discussed in section 4.4.2). This planner defines a simultaneous action as: both parts turning about their part-axles in opposite directions such that the unconnected part's face direction does not change as shown in Fig. 4.15.

[21]Definition: *forwards* turning for actions 0-3 and 10-13 is defined any angle that increased the rotational position of the part about its own part-axle. This in turn is defined as so: A part is at 0 degrees if the part's face is aligned with the central pin (Fig. 4.3), it is +90 degrees if the part's face is point in the position X/Y/Z direction, and -90 degrees if pointing in the negative X/Y/Z direction

Table 4.1: Action Index

Code	Action Name	Description
-2	fully connect	all connectors of a module to form connections if possible
-1		the *null* action
0	inc unconn 90 deg	unconnected part rotates forwards about its axle by 90 degrees
1	dec unconn 90 deg	unconnected part rotates backwards about its axle by 90 degrees
2	inc conn 90 deg	connected part rotates forwards about its axle by 90 degrees
3	dec conn 90 deg	connected part rotates backwards about its axle by 90 degrees
4	inc central 90 deg	parts twist anticlockwise from connected part's perspective
5	dec central 90 deg	parts twist clockwise from connected part's perspective
6	simultaneous action	both parts rotate, unconnected part's face direction is unchanged
7	switch connection	unconnected part connects, then connected part disconnects
8	connect/disconnect conn from helper	connected part toggles its connective status with helper
9	connect/disconnect unconn from helper	unconnected part toggles its connective status with helper
10	inc end 90 deg	helper's end part rotates forwards about its axle by 90 degrees
11	dec end 90 deg	helper's end part rotates backwards about its axle by 90 degrees
12	inc base 90 deg	helper's base part rotates forwards about its axle by 90 degrees
13	dec base 90 deg	helper's base part rotates backwards about its axle by 90 degrees
14	inc helper central 90 deg	helper parts twist anticlockwise from base part's perspective
15	dec helper central 90 deg	helper parts twist clockwise from base part's perspective

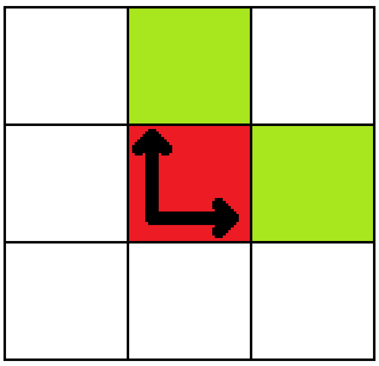
4.5.7 Collision Detection

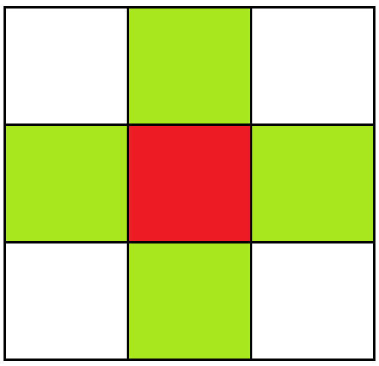
collisionDetection (state, action) $\longrightarrow$ (Boolean) willCollide

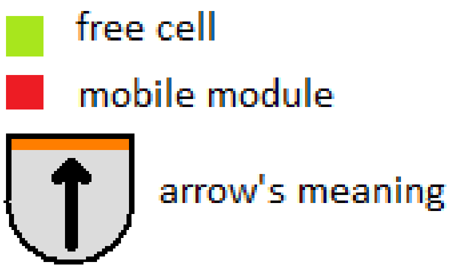
Collision Detection answers the following; if a module were in a certain state, and it took a certain action, would this module collide with another module? This form's part of the planner's *Transition Model* and is required when expanding state-nodes to verify which action-links are safe.

Early Routine: ***Hard Coded***

Initially *Collision Detection* was purely hard-coded, when the planner only planned for single module movements. The function was a big switch-case routine that took different combinations of state & actions and matched them to a set of cell positions relative to a module's state that the module would encroach on when taking a certain action. These encroached cells would all need to be vacant (no modules currently located in them) for an action to be valid, otherwise the module would collide with another there. There were originally 7 moving-actions a module could take (actions 0-6 as defined in sec. 4.5.6). Actions 0-1 rotate the unconnected part either forwards or backwards 90 degrees and require two vacant cells as shown in Fig. 4.16a. Twisting actions of the central pin (actions 4-5) requires four vacant cells around the unconnected part that will be the one moving, as seen in Fig. 4.16b. For actions involving the connected part, turning it 90 degrees (actions 2 & 3), the cells that need to be vacant around it are dependent on the module's part axles as well, if they are aligned or not. The vacant cell requirements for these actions are shown in Figs. 4.17 & 4.18. There are 3 red squares indicating the module connected part, and the unconnected part's start and finish cells. Lastly, the simultaneous action is also shown in Fig. 4.19. The benefits of this actions were discussed in sec. 4.4.2 and are seen here.


(a) Actions 0 & 1: unconnected part turning 90 degrees


(b) Actions 4 & 5: twisting central pin



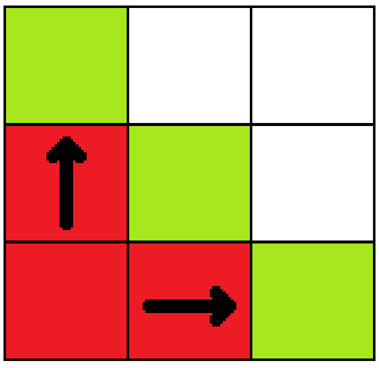
(c) Legend

Figure 4.16: Unconnected part actions (*view along spin axis*)

Notice how an unconnected part that is facing upwards in the bottom-middle cell can move upwards to connect its face connector to whatever module may be located in the top-left cell. Looking at Fig. 4.17a, a module trying to orient itself in the same way without simultaneous actuation encroaches on the top-left cell, and thus would collide with any module before it has a chance to connect.

Current Routine: *Almost hard-code free*

Currently, *Collision Detection* is mostly automated. A geometrical routine was written that takes a group of modules and an axis of rotation, and records all the cells encroached on by each module in the group as the whole group swings though 90 degrees in a given


(a)

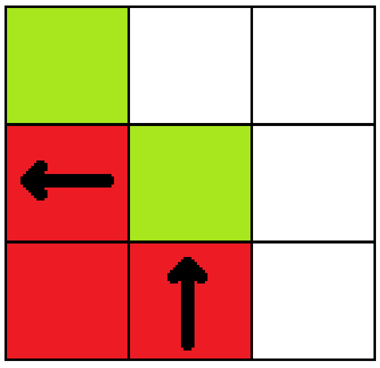

(b)

Figure 4.17: Actions 2 & 3: connected part turning 90 degrees: parallel part axles (*view along spin axis*)

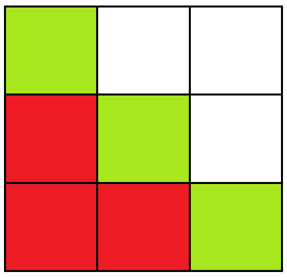


Figure 4.18: Actions 2 & 3: connected part turning 90 degrees: perpendicular part axles (*view along spin axis*)

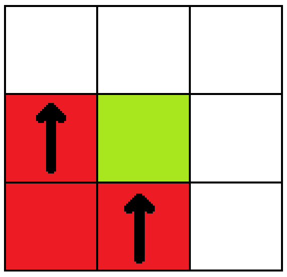
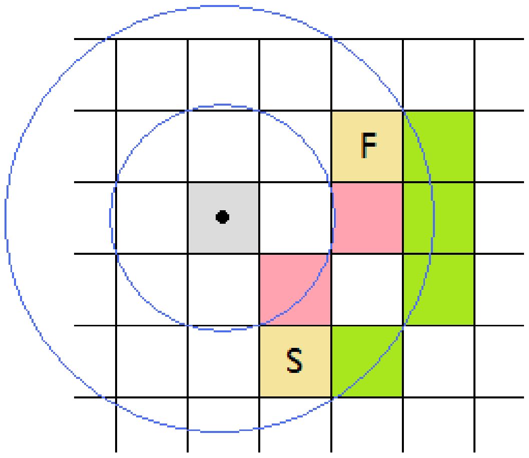
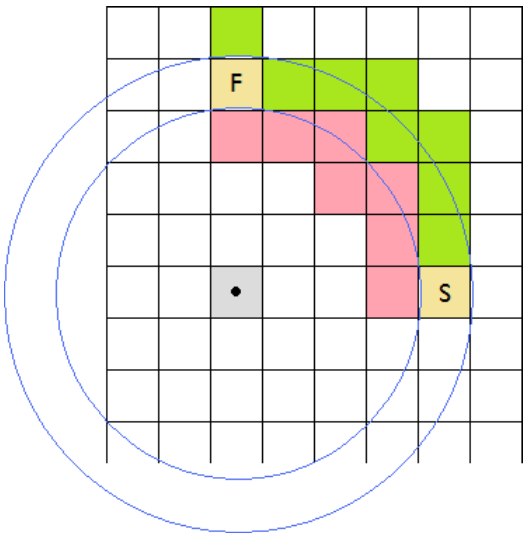


Figure 4.19: Action 6: both parts turn simultaneously, both axle parts must be parallel (*view along spin axis*)



(a) encroached cells; example from a part not in-line with center of rotation



(b) encroached cells; example from a part in-line with center of rotation

Figure 4.20: Collision detection: encroached-cells from one part's transition though space by 90 degrees, starting at cell 'S' and finishing at cell 'F'. The dotted cell represents the center of rotation

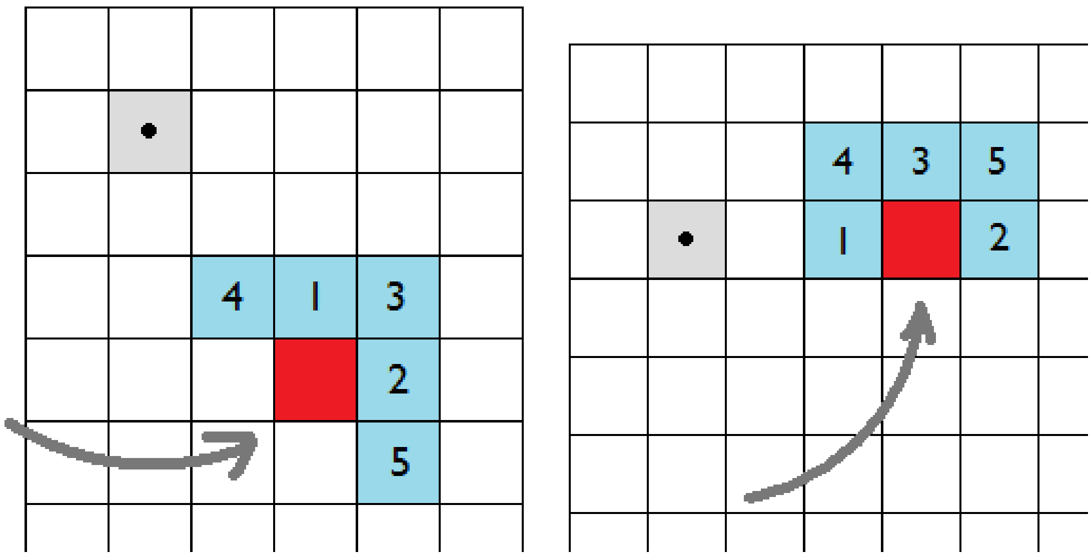
direction. This is done by locating the nearest and furthest point on each module part's surface from the center of rotation. It then follows radii out from these points (about the center of rotation) for 90 degrees, and lists all the cells that have vertices within the min radii - max radii section. Cells with vertices within this section will either be partially or fully encroached on, but encroached on nevertheless so must be vacant for a particular action to be marked safe and valid. Fig. 4.20 shows two examples of the cells encroached on by one part.

This automation needed to be included when helpers were introduced to this thesis, the prospect of hard-coding all state/action combinations to map to encroached-cell lists was too daunting. It is also part of this thesis ultimate goal, to be able to plan reconfigurations for arbitrary module designs. To do so, *Collision Detection* must be fully automated without relying on any hard-code. The only hard coded parts in *Collision Detection* is to determine the cells at the start and finish regions of these min/max radii lines drawn out. The planner does not account for certain curved surfaces of the modules, and so wrongly marks some cells as encroached and wrongly marks some cells as not encroached in these locations. Hard coding specific to the module's design fixes up this small number of 'mistakes' the autonomous part makes (algorithm 4.6 lines 23-27). Automating this last section of hard code is possible, but complex[22] and will take time.

Algorithm 4.6: ***Collision Detection***

Collision Detection begins by calculating general information about a *swinging-arm* of modules, such as center of rotation, the plane it is spinning on and the rotation direction (lines 1-3). It must check every module's part that make up the module's arm to check if it will collide with something (line 4), and so this routine can be executed on an arbitrary number of connected modules even though this planner only ever uses it for just 2 modules (a helper and a mobile module). For each part, a several radius arcs are 'drawn' out to discover which cells gets encroached on (as seen in Fig. 4.20. To search this a '*current-cell*' is initiated at the part's start position (line 9), and tracks this radius curve. By tracking this curve, the current-cell is always a cell that the part will encroach on, thus if a module part is there, a collision would occur, and the routine can return (lines 13-18). Obviously if the part there belongs to the swinging arms of modules, this will not collide as both module parts will move as one so this needs accounting for (line 15). To 'track' this radius, and once the current cell has checked no part exists in its location (line 15), it searches

[22]it requires a standardized way of representing arbitrary modules geometrically, such that when the planner changes to a different module type, a 'geometric description' can be passed to Collision Routine to autonomously derive all encroached cells. The 'complexity' will be in developing both a 'standardized geometric description' and an extra subroutine in Collision Detection to make sense of this

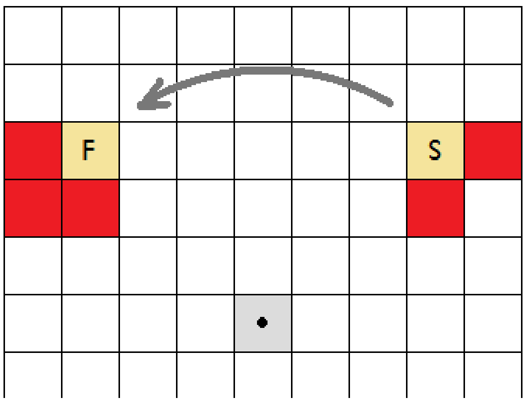


(a) try-cells from a cell not in-line with center of rotation

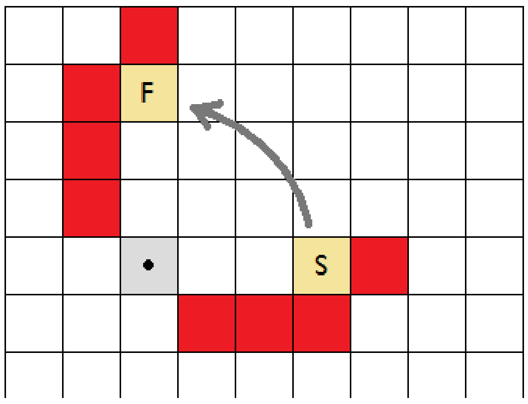
(b) try-cells from a cell in-line with center of rotation

Figure 4.21: Collision detection: try-cells. From collision detection search's current cell it has checked, it will progress to one of these try-cells in an order as shown by the numbers to find the next cell that this part will encroach on. (A part starts at cell 'S' and finishing at cell 'F'. The dotted cell represents the center of rotation)

neighboring cells for the next 'current-cell' to check. As the radius is a continuous line, there will always be a cell either adjacent to or diagonal to the current-cell which the radius intersects. The current-cell tests out *try-cells* to see which one this is, in a particular order as shown in Fig. 4.21. This order is a result of testing to code, to ensure a search does not go backwards. It also ensures a search does not skip over a cell, as it is possible that more than one try cell will be encroached on, and both can still be 'current cells' and check for collidable objects if done so in the correct order, but if not done in the correct order the 'more forward' try-cell could be selects which will progress the search onwards and forget to consider the cell it left behind.



(a) forbidden-cells from a cell not in-line with center of rotation



(b) forbidden-cells from a cell in-line with center of rotation

Figure 4.22: Collision detection: forbidden-cells. These marks the beginning and end of an ‘encroached cell search’ that the search should not go beyond. (A part starts at cell ‘S’ and finishing at cell ‘F’. The dotted cell represents the center of rotation)

Forbidden cells (Fig. 4.22) are placed in order to indicate to a search when to stop cell-tracking, and also as an extra precaution that the search does not go backwards when current-cell begins at the starting part position ‘S’. No try-cells that are also forbidden cells are considered (line 13) and when the next current-cell is the final position the part will end up in after a 90 degree turn (line 10), the search for this particular part completes, and another starts (line 4).

Different parts of the part will collide will be responsible for encroaching on different cells, as seen by the different encroached cell shading of Fig. 4.20. Therefore each part needs to have several radii search from different parts of its geometry to work out every cell possible that be encroached on (line 8). This includes a radius made by the closet and by the farthest points on a part’s shape. Sometimes this is enough as seen in Fig. 4.20b, but it can occasionally miss the odd cell as seen by the white cell in between both radii in Fig. 4.20a. To account for this a ‘center radius’ is drawn out from the module part’s position as well (line 8). As mentioned, some hard coded cells are checked at the routines end, that fix up ‘mistakes’ made by this routine just described, by adding in some more cells to check that are dependent on module geometry. At the moment this hard code is specialized to the Superbot module sec. 2.3.5. Finally if no cells that will be encroached on contained modules the swinging arm will collide into, this routine return ‘false’ (line 28).

4.5.8 Translation

translation (module, state-defined action) $\longrightarrow$ (Action) configuration-defined action

As discussed in sec. 4.2.2, there can be many configurations to a single isomorphic state of a module, and to execute a movement, the planner will (at the last minute) require a module’s full configuration description to perform an action. For example, if a Superbot module (sec. 2.3.5) were to turn one of its parts 180 degrees, it is still in the same state, as it functionally equivalent to before, yet the actuator polarity that drives the part’s face forwards/backwards +/- 90 degrees will be reversed. If configuration is not taken into account, the actuators could perform the opposite action to what is desired due to lack of information. This is simply a matter each module keeping a log of its own configuration, not to be searched, but for these last minute action-translation from state-defined actions to configuration-defined actions.

Algorithm 4.6 Collision Detection

```
1: get center of rotation
2: get direction of spin axis // (X or Y or Z)
3: calculate spin direction // (clockwise or anticlockwise)
4: for each module Part in Swing-Arm do
5:     get Start-cell of Part
6:     get Finish-cell of Part
7:     get Forbidden-cells
8:     for Radius = [Closest, Center, Farthest] surface points of Part to center of rotation
       do
9:         Current-cell $\leftarrow$ Start-cell // (initialize)
10:        while Current-cell $\neq$ Finish-cell do
11:            get next Try-cells based on Current-cell
12:            for each Try-cell do
13:                if Try-cell is intersected by Radius AND Try-cell $\neq$ a Forbidden-cell then
14:                    Current-cell $\leftarrow$ Try-cell // (update)
15:                    if Current-cell contains a part that does not belong to Swing-Arm then
16:                        return True: 'part Part will collide with cell Try-cell'
17:                    end if
18:                end if
19:            end for
20:        end while
21:    end for
22:    for all hard-coded Cells to check near Start-cell and Finish-cell do
23:        if Cell contains a part that is not in Swing-Arm then
24:            return True: 'part Part will collide with cell Cell'
25:        end if
26:    end for
27: end for
28: return False: 'no collisions will occur. Every encroached cell does not contain a part
    not of Swing-Arm'
```

4.6 Evaluation

This section evaluates the CSRP in terms of its performance and capabilities. An analysis explores CSRP properties which follows with some example reconfigurations currently achievable by the CSRP. Being a *deterministic* software algorithm; testing does not play as significant a role as for stochastic planners because it either will perform a certain reconfigurations or it will not. Nevertheless it is still of interest to ascertain the competency of the CSRP; what subset of reconfigurations are currently achievable, and also to determine which design goals have been met with a study of relevant reconfiguration statistics compiled for each example.

Overall, the CSRP has shown that planning modular reconfigurations in native kinematics is efficient; allowing a planner to reduces the size of the state space substantially by exploiting module lattice structure and symmetries as much as possible. It also maximizes the use of module functionality, considering all actionable options possible. This is in contrast to generic planners designed to be portable between hardware types at the expense of using a module's complete functionality or planners that restrict themselves to reconfigurations of metamodules in order to simplify the search process. Ultimately a good SRP should take full advantage of the hardware at its disposal, and native kinematic planners like the CSRP do this.

4.6.1 Analysis

This centralized planner written in Java can autonomously reconfigure Superbot-based robots between arbitrary shapes via serial locomotion of motions. The CSRP is tested in simulation using Java3D as a visual debugging tool using a Superbot simulator written by David Brandt from the ISI. Time complexity of computing a reconfigurable solution is at most quadratic polynomial with the number of modules in the system. This quadratic relationship prohibits scalability of this algorithm, however all centralized planners are faced with the problem of having higher-than-linear time complexity and thus are always unsuitable choices for modular robots. What this planner successfully prototypes is a method of planning in native kinematic space achieving close-to-optimal robot reconfigurations, defined as the least number of atomic actions required needed to reconfigure. A decentralized version of this planner is scalable and discussed in chapter 5.

Memory Usage

Memory usage is an important consideration in modular robotics as each module has a limited memory capacity from which to allocate to non-routine computational tasks like reconfiguration planning. Using today's off-the-shelf memory devices, modules can be fitted with several SD memory cards regularly used in personal cameras supplying memory capacities of several gigabytes per module [3]. In many cases, planning with this amount of memory is more than adequate. Exceptions include large lookup tables for modules to refer to in a variety of situations where certain action sequences can be stored as macros, or planners that use elements of artificial intelligence that prefer ever-growing knowledge bases to optimize their decision making.

The principal usage of memory is Local Search's creation of state networks, specifically the *states* within the state network. Super states require 756 bytes each, of which most mostly comprises the mobile module state (opposed to the helper module states which are indexed due to high usage) at 704 bytes out of the 756. The reason for this cost is the class doesn't just hold values of primary variables that define it (listed in sec.

4.3), it also holds supplementary variables that help calculate neighboring states and other frequently-accessed information such as connector locations and pointers to neighboring states etc. Worst case scenarios included every structural module is a double-helper (where both module parts have the capacity to be the helper end or base part) contributing 19020 states to the state network. This however does not boost memory requirements by $19020 \times 756 bytes = 13.7 MB$ as all mobile module states (the memory expensive part of 704 bytes) are shared by storage in a hashed set which multiple super states point to[23], thus saving the duplicated work and storage of the supplementary information accompanying a state. There can only be 2808 mobile module states within the proximity of each helper, so worst case scenario is $19020 \times (756 - 704) bytes + 2 \times 2808 \times 704 bytes = 4.71 MB$. This is not severe, memory requirements increase linearly with the amount of modules present, so a 2GB-RAM computer used to simulate these reconfigurations could potentially solve reconfigurations of 434 Superbot modules[24], however in practice the Java3D Superbot simulator limits this to approx 55 modules by reserving memory for itself for the 3D modeling and graphical output.

Execution Time

Time complexity of the CSRP is quadratic polynomial with the robot's number of modules, which is the best relationship a centralized planner can expect to achieve, and exampled in section 4.6.2. Reconfiguration simulations have been conducted with up to 20 modules which incur execution times in the order of several minutes using a single 2GHz processor.

[23]for example if a certain superstate involves a mobile module state and a disconnected helper module state, the helper module could move which would change the superstate. However the mobile module state itself has not changed (the mobile module did not move) so its information should not have to be recalculated, instead both these superstates independently point to that particular mobile module state

[24]$2GB/4.71MB \approx 434$

There are several main factors that determine execution time. One is clearly the dissimilarity between a robot's initial and final configurations; if both shapes are very similar then only a few modules need relocating and a solution would be found in a small amount of time. Conversely extremely dissimilar configurations will require the relocation of most modules, each relocation warranting the creation of a new state-network which is relatively time expensive. A second factor in execution time is the number of helper modules present. As mentioned in section 4.4.1, helper modules are the biggest contributors to a state-network's size. The amount of helper modules in a robot cluster is dependent on its configuration; helper modules must have at least one part that can detach and move about without violating global connectivity. In the current CSRP implementation all modules which be verified to do this (using *Connectivity Checker*; sec. 4.5.4) that are not already classed as mobile modules are thus classed as helper modules. This increase in state-network size necessitates more planning time from the CSRP as seen later in table 4.2. A possible improvement on the CSRP is to limit the amount of module classed as helpers. This would remove their ability to help mobile modules but many helpers never get the chance to help anyway. If a more sophisticated method can be developed that determines which potential helper modules will be used and which will not then a significant amount computation could be spared.

The CSRP spends most of its time within the *Local Search* (sec. 4.5.5) routine, linking up state networks after they have been populated, particularly in the *Collision Detection* routine (sec. 4.5.7) comprising 55% - 75% total execution times (depending on the reconfigurable shape). Even though the quadratic relationship of the CSRP is unavoidable, execution time can potentially be scaled back by further optimizing *Collision Detection*. This could be done by tabulating the encroached cells [25] for all possible state-action pairs which would only be a once-off mass computation. Referencing this instead of running Collision Detection again could cause a 2 to 4 fold reduction of execution time (based on the 55% & 75% values) depending on the reconfigurable shape. Such a 'mass computation' would required the examination of over 101,000 possible superstates each of which can perform up to 17 possible actions. This computation can be independently computed and stored in a data file for all future CSRP simulation to access and works out to be roughly $145MB$ in size[26]. Whilst this may not be a problem for desktop computers conducting CSRP simulations, one must ensure modules had adequate free memory to each store the $145MB$ table to take advantage of this under the DSRP implementation.

Correctness & Completeness

The CSRP is a *correct*[27] program however is not a *complete*[28] one. Correctness has been independently tested with the Superbot simulator software which is used to provide visuals[29] of robot reconfigurations directed by the CSRP. The simulator uses the Java3D engine to throw errors if simulated 3D objects (modules in this case) pass through each other, which is a collision in the real world and would violate program *correctness*. The second factor of correctness; maintaining global connectivity is verified independently by the Superbot simulator itself (not the Java3D engine) which requires global connectivity of any simulation,

[25]is first computes any cells a particular move will encroached on (a time consuming task as discussed in sec. 4.5.7), and then checks if any of those cells currently contain other modules

[26]calculated using the average number of encroached cells each state-action pair produce (3.53), and the memory required by a java virtual machine to store Point3i (3D location) objects

[27]*correct* is defined by; if a program finds a solution, then that solution is correct. A correct reconfiguration means no collisions and no global disconnections

[28]*complete* is defined by; if there is a solution to be found then the program will find it

[29]such as the figures throughout section 4.6.2

and will throwing an exception if this occurs. CSRP has its own routines prevent either collisions or global disconnections; *Collision Detection* and *Connectivity Checker* (sec. 4.5.7 & 4.5.4), however the Superbot simulator and Java3D simulations have confirmed CSRP correctness independently.

Completeness has not yet been achieved. The CSRP can handle a large subset of arbitrary reconfigurations however fails in more complex cases which demand the temporary use of some modules as ‘scaffolding’ that others modules use to climb over. Currently the CSRP has no way of dealing with *temporary* relocation of modules for the packing benefit of others. Such a task presents a host of new challenges including ensuring that no scaffolding-type modules get stuck in situ by surrounding modules when they next get called to relocate. The thesis work by Ben Itzstein and Michael West mentioned in the preface extends Million Module March’s capabilities to handling the temporary placement of some cubes for the sake of robot stability, but to the authors knowledge no existing SRP yet deals explicitly with the challenge of the intentional use of ‘scaffolding-modules’ in an intentional fashion for the purpose of reconfiguring otherwise non-reconfigurable shapes. One example of this is the ‘line-to-ring’ example in section 4.6.2. While this reconfiguration is possible with 6 modules assuming helper modules have sufficient torque to carry only one mobile module at a time, larger single-file ring shapes are impossible to construct without any scaffolding behavior. Easier reconfigurations include ‘line-to-box’ types which are possible for any number of modules owing to their solid, non-sparse geometries.

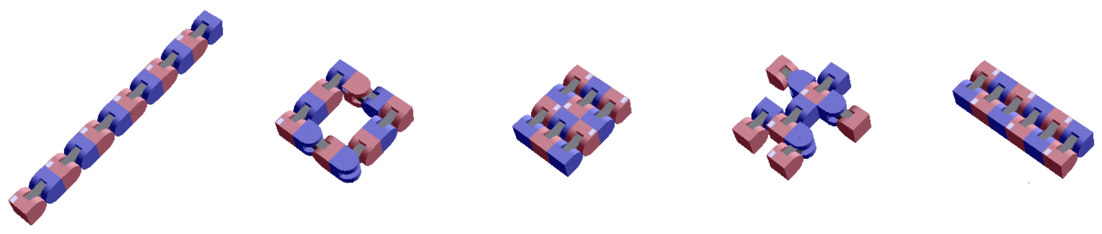

(a) Line (b) Ring (c) Box (d) Superbotman (e) Sidestack

Figure 4.23: Shapes chosen for reconfiguration examples

4.6.2 Examples

This section examines several CSRP-directed reconfigurations of simulated Superbot modules. The examples discussed are also available in video format[30] located in the attached DVD. Selected shapes for this section are a line, ring, box, stick figure (‘Superbotman’) and a side-stack seen in Fig. 4.23. Reconfigurations examples are been between the following shapes (tested in both directions):

- Line $\longleftrightarrow$ Ring
- Line $\longleftrightarrow$ Box
- Line $\longleftrightarrow$ Superbotman
- Line $\longleftrightarrow$ Sidestack

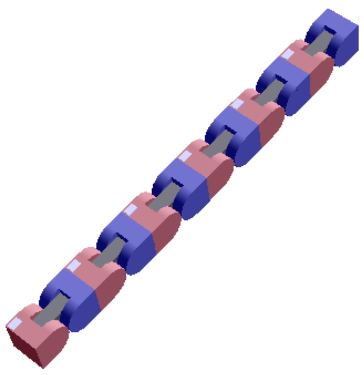
Perhaps the most interesting reconfiguration is the *line-to-ring* example seen in Fig. 4.24. In this example the CSRP used 4 iterations of Global Search resulting in 4 successive module relocations shown in Figs. 4.24b, 4.24h, 4.24i and 4.24j. Figs. 4.24b through 4.24h show seven snapshots of the second relocation (the mobilized module is the bottom-left module in Fig. 4.24b) involving cooperative behavior. The running value of ’steps’

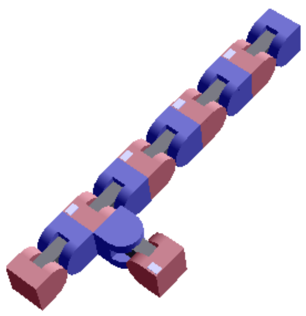
[30]an index of all multimedia is located in appendix A

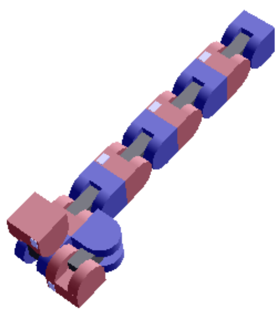
is the total states transitioned through robot's state space. This includes regular atomic actions taken from the initial reconfiguration to that have physical significance such as movements and connections, and also actions dedicated to planner protocol such as helper-engagement[31]. In this particular relocation, the CSRP's found it was optimal to use a helper module to pick up the mobile module (step 20), reorient it (steps 21-29), and place the mobile module back down in its original location (step 30). Interestingly the CSRP found this technique to be optimal because from step 34 it was able to re-grip the mobile module in such a way they were both in their correct configurations (stipulated by Global Search) by step 38. The second half of the ring, which is symmetric to the first half created by step 38, did not need to follow an analogous action sequence. This is because the CSRP was then able to take advantage of the fact the first half was already built, and so from step 55 onwards, a helper module could place the final mobile module (top module in Fig. 4.24i) onto the existing first half of the ring for attachment and immediately release and re-grip without violating global connectivity.

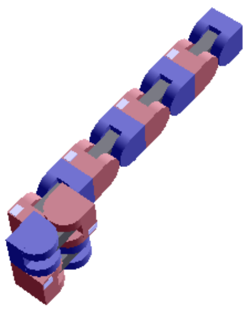
The CSRP is not yet able to guarantee reverse-reconfigurations. In practice a robot can always log its actions and perform them in reverse order to return to a pre-existing state, however in the absence of a log; CSRP will sometimes choose a different return strategy. The problem with this is some shape changes are only one-way. For instance the CSRP can direct a robot to transform from the line shape into a ring but fails when attempting to reconfigure the ring shape back into a line. This shortcoming is due to Global Search's selection of the 'wrong' module to mobilize on the return journey. Instead of mobilizing

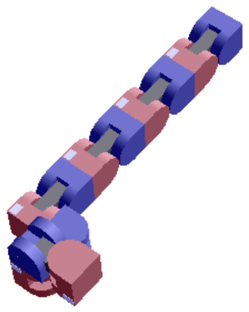
[31]when a mobile modules calls upon a helper-related action, the CSRP must be able to distinguish which helper module the mobile module is referring to. This takes place with the use of engagement-actions outlined previously in sec. 4.5.6. These actions are purely in software and because they impose no physical ramifications to a robots the 'steps' should not be considered as a metric for comparison with other planners but nevertheless indicate the path distance through state space traversed by the robot

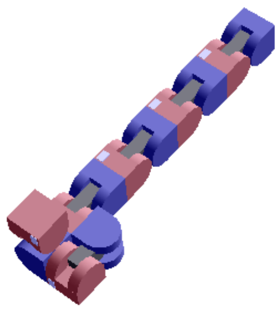

(a) step 0

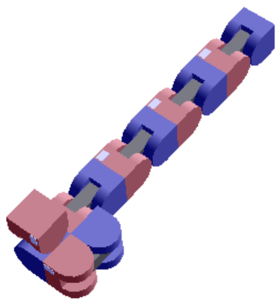

(b) step 15

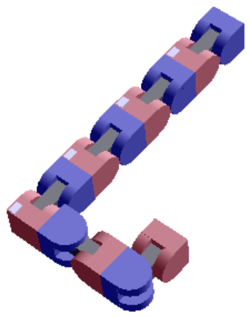

(c) step 20

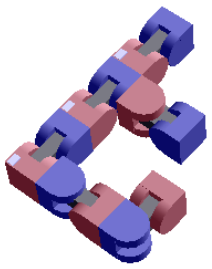

(d) step 23


(e) step 29


(f) step 30


(g) step 34


(h) step 38


(i) step 55

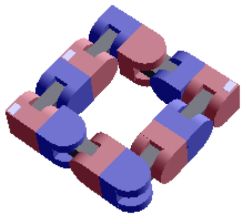
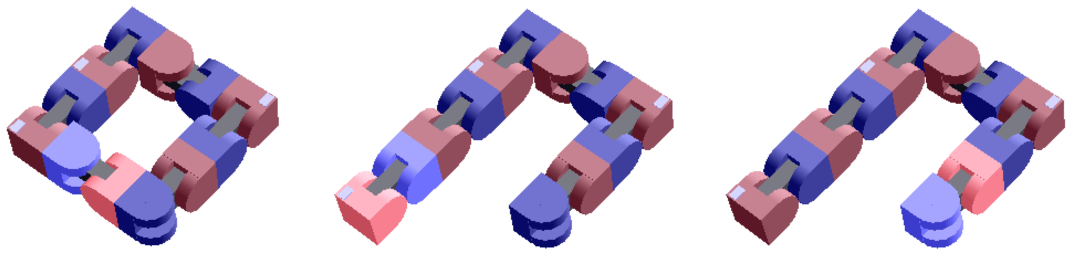

(j) step 65

Figure 4.24: Reconfiguration example: Line-to-Ring



(a) step 0: first mobile module is selected

(b) step 11: first mobile module has relocated

(c) step 11: second mobile module is selected

Figure 4.25: Failed reconfiguration example: Ring-to-Line

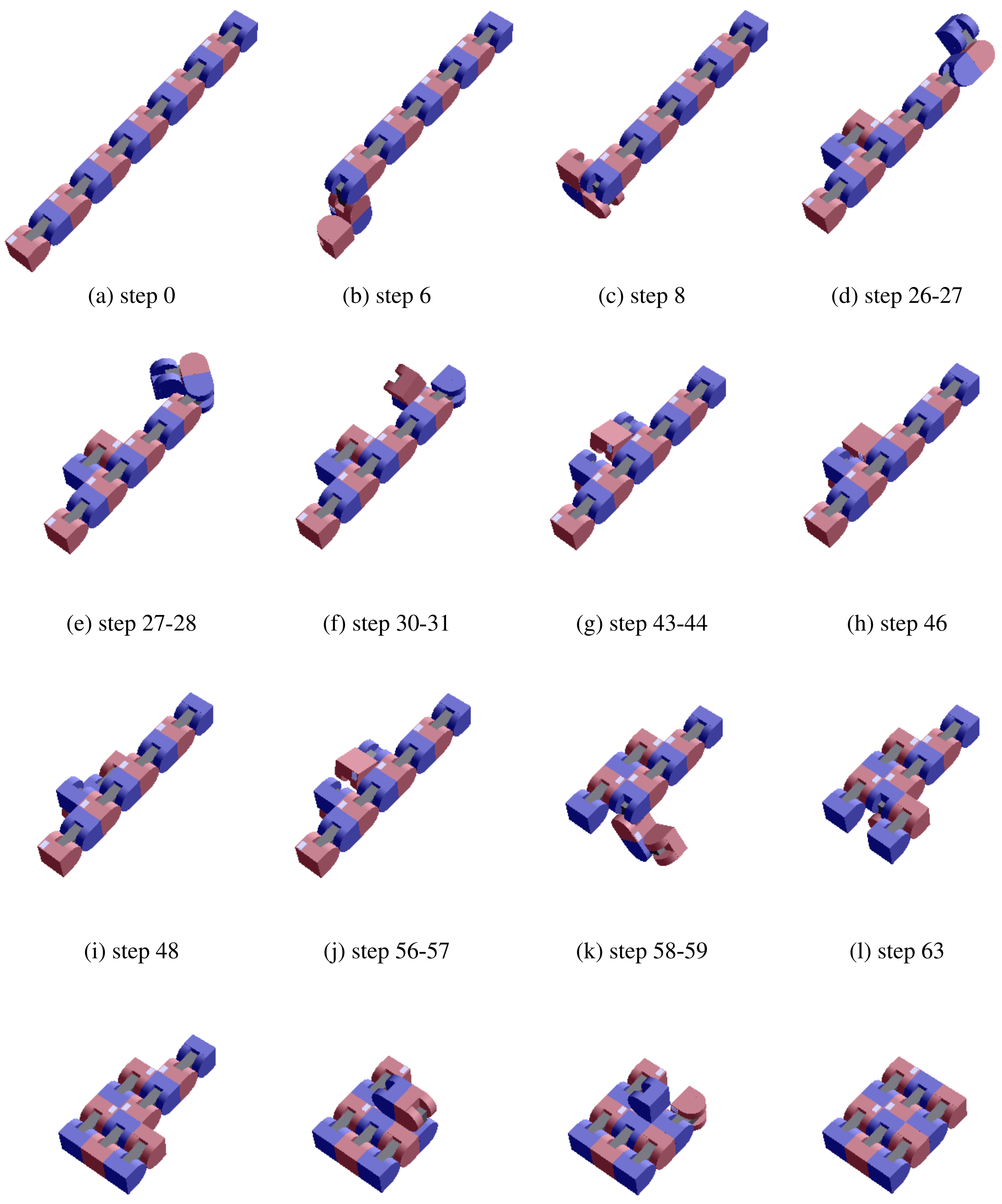
the last mobile module in the previous example (Fig. 4.24) to retrace its steps it mobilizes the first (lighter colored module Fig. 4.25a), leaving the ring structure three-quarters intact by Fig. 4.25b (Modules chosen to become mobilized are those which are (a) not in a goal position yet and (b) can safely detach without violating global connectivity). The sole choice of which module to mobilize now by Global Search is the lighter colored module in Fig. 4.25c. CSRP soon returns a fail at this point this module cannot traverse leftwards, even with coordinated help from any one other module, where Global Search attempts to reconstruct the line.

Complexity of reconfigurations varies widely with the shapes to be reconfigured, and is often anti-intuitive. The configuration between a line to a sidestack, both one-dimensional shapes, has at least a threefold execution time over all other reconfigurations from a line. Planner performance statistics for each example is recorded in tables 4.2 and 4.3 for comparison. Performance parameters included:

Execution Time: seconds CSRP took to execute on a 2GHz laptop

Memory Required: the maximum amount of memory used during runtime

Superstates (max): the maximum number of superstates nodes expanded by Local Search during an iteration of Global Search



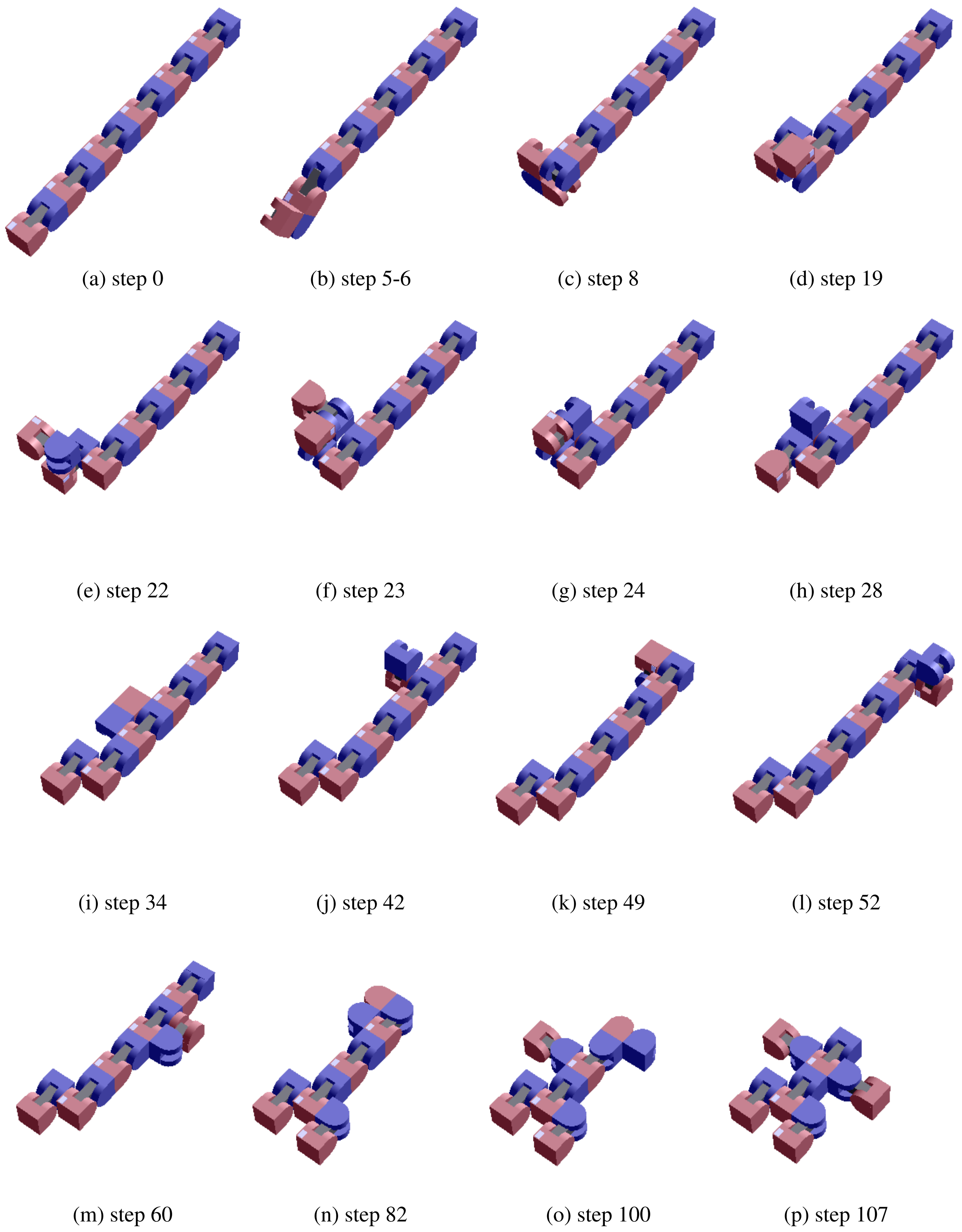
(a) step 0 (b) step 6 (c) step 8 (d) step 26-27

(e) step 27-28 (f) step 30-31 (g) step 43-44 (h) step 46

(i) step 48 (j) step 56-57 (k) step 58-59 (l) step 63

(m) step 66 (n) step 76 (o) step 78 (p) step 83

Figure 4.26: Reconfiguration example: Line-to-Box



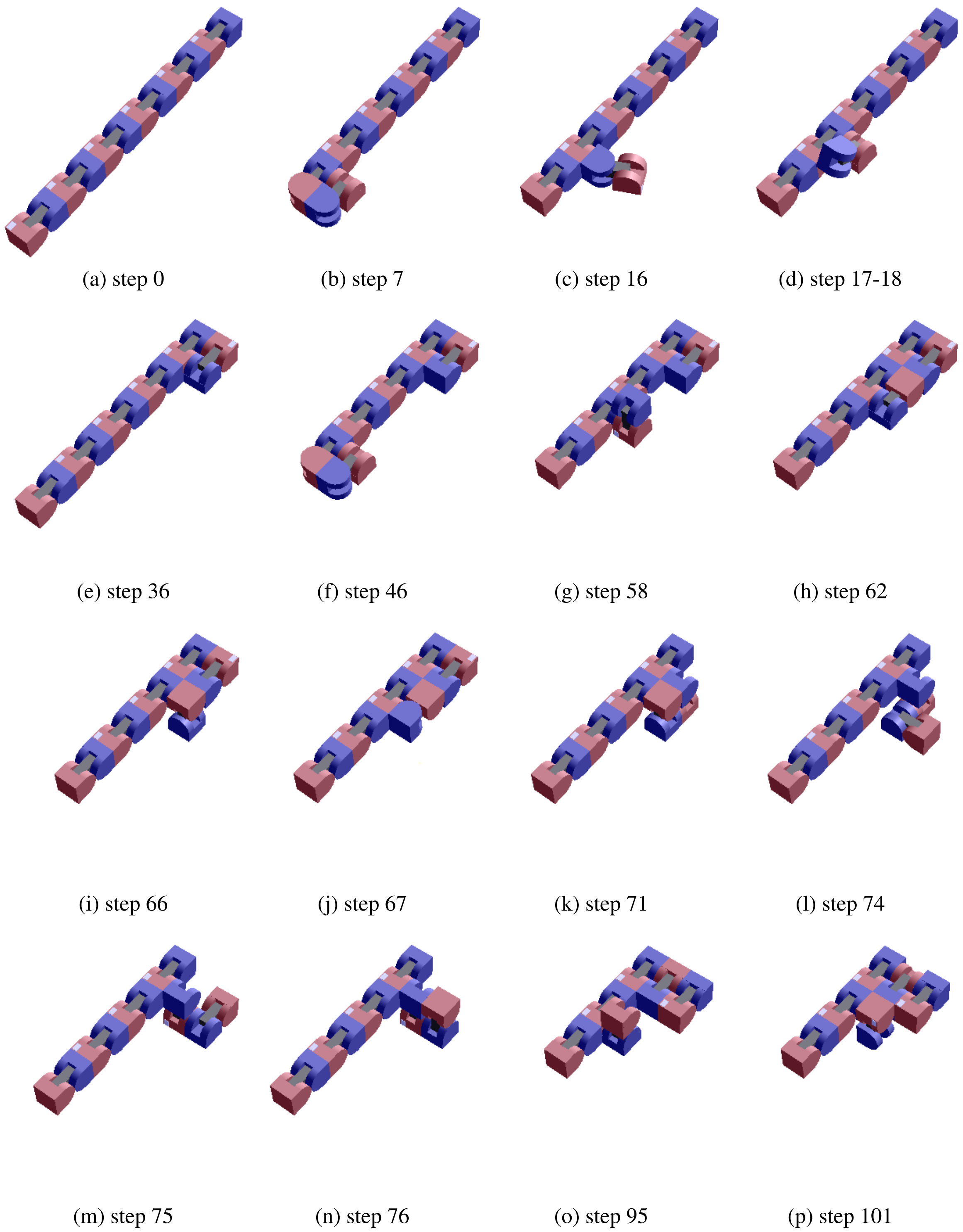
(a) step 0 (b) step 5-6 (c) step 8 (d) step 19

(e) step 22 (f) step 23 (g) step 24 (h) step 28

(i) step 34 (j) step 42 (k) step 49 (l) step 52

(m) step 60 (n) step 82 (o) step 100 (p) step 107

Figure 4.27: Reconfiguration example: Line-to-Superbotman

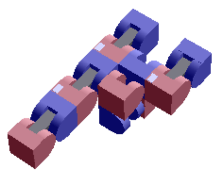

(a) step 0 (b) step 7 (c) step 16 (d) step 17-18

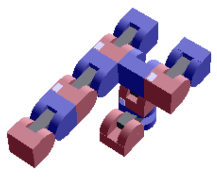
(e) step 36 (f) step 46 (g) step 58 (h) step 62

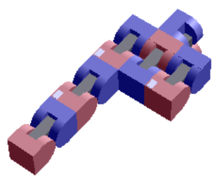
(i) step 66 (j) step 67 (k) step 71 (l) step 74

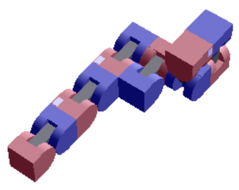
(m) step 75 (n) step 76 (o) step 95 (p) step 101

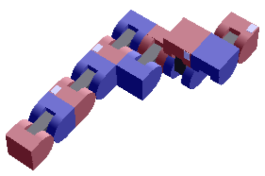
Figure 4.28: Reconfiguration example: Line-to-Sidestack (Part 1 of 2)

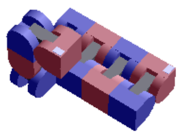

(a) step 104

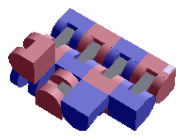

(b) step 105

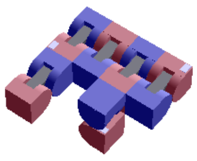

(c) step 107

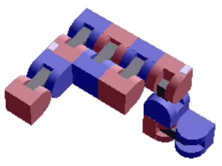

(d) step 113

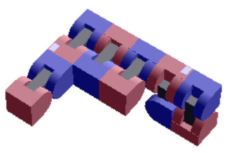

(e) step 116

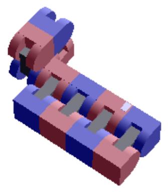

(f) step 124

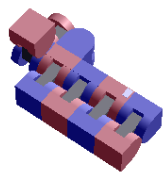

(g) step 127

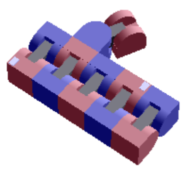

(h) step 132

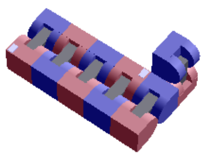

(i) step 139

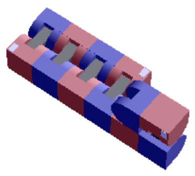

(j) step 140


(k) step 153


(l) step 156


(m) step 160-161


(n) step 170


(o) step 174

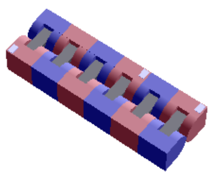

(p) step 179

Figure 4.29: Reconfiguration example: Line-to-Sidestack (Part 2 of 2)

Superstates (total): the total number of superstates nodes expanded by Local Search

Actions (connections): total number of connector-based actions performed

Actions (unaided motion): total number of locomotion actions performed by a mobile module

Actions (helper motion): total number of locomotion actions performed by a helper module

Actions (total): sum total of the above three action types

Max Search Depth: the maximum breadth first search depth by Local Search during an iteration of Global Search

Max Helpers: the maximum number of modules classed as helpers by Global Search during an iteration of Global Search

Table 4.2: Reconfiguration Statistics Compiled for CSRP Examples of 6 modules

Reconfiguration	**Line to Ring**	**Line to Box**	**Line to Superbotman**	**Line to SideStack**
Execution Time (s)	9.93	21.33	13.17	64.12
Memory Required (MB)	4.68	12.06	6.93	12.19
Superstates (max)	10,327	42,716	17,519	87,365
Superstates (total)	36,680	88,472	45,022	263,148
Actions (connections)	15	16	24	39
Actions (unaided motion)	14	32	52	76
Actions (helper motion)	30	26	22	46
Actions (total)	59	74	98	161
Max Search Depth	23	30	58	39
Max Helpers	2	10	4	10

The reason for prolonged executions of reconfigurations involving the sidestack shape is due the higher number of modules designated as helpers. As tables 4.2 and 4.3 record,

Table 4.3: Reconfiguration Statistics Compiled for Reversed CSRP Examples of 6 modules

Reconfiguration	**Ring to Line**	**Box to Line**	**Superbotman to Line**	**SideStack to Line**
Execution Time (s)		11.34	26.31	50.51
Memory Required (MB)		12.22	8.09	12.19
Superstates (max)		22,738	138,409	90,118
Superstates (total)		45,431	154,997	255,258
Actions (connections)	*failed*	17	16	40
Actions (unaided motion)		34	36	80
Actions (helper motion)		24	20	42
Actions (total)		75	72	162
Max Search Depth		29	38	40
Max Helpers		10	5	10

this reconfiguration had 10 helper modules[32] during a relocation. As mentioned; helper modules increase the size of the searchable state network significantly, a fact which is supported by the many superstates expanded during the search for sidestack's reconfigurable solution. Nevertheless, reconfigurations involving the box shape also had 10 helpers during an iteration of Global Search (module relocation) and they executed relatively faster. This discrepancy results from the different placements of helper modules about any mobile module, characterized by the initial & goal shapes chosen. As seen in Fig. 4.26 the box example involves four relocations and each time the mobilized module is not required to move far. This means as Local Search (which is a BFS-based), spreads outwards through the large state space (which is similar in size to that of sidestack's) it discovers the goal state (to relocate the module) relatively early. This fact is supported by the values of 'Maximum Search Depth' in each box/sidestack example (tables 4.2 and 4.3) which are of 30/39 and 29/40 respectively. Local Search in the sidestack example is undeniably searching deeper before it finds each goal state. Figures 4.28 and 4.29 shows why sidestack-reconfigurations

[32]there are only 6 modules total, but 5 of these modules are double-helpers

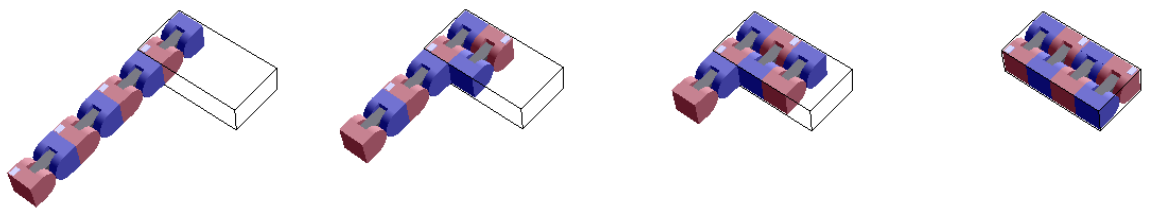

(a) step 0 (b) step 25 (c) step 45 (d) step 72

Figure 4.30: Successive relocations: Line-to-Sidestack, 4 modules

involve deeper searching. The only choice of modules Global Search has for mobilizing is the bottom-left module shown in the sub-figure of each new iteration[33]. The goal location of each successive mobilized module to traverse to the rightmost vacancy of the sidestack by virtue of the user-inputted shape. Therefore every mobilized module is traversing from end-to-end of the robot, and so starting/goal states Local Search attempts to link are also approximately located at opposite ends of the state space. Thus the CSRP is essentially searching the entire state network for every iteration of Global Search. These state network are usually very large in size at well (one recorded as 90,118+ states) because every module added to the sidestack is classed a double-helper for all subsequent iterations of Global Search[34]. Thus the sidestack-reconfigurations result in the expansion of many more states that other examples and thus higher execution times.

CSRP's time complexity is quadratically polynomial to the number of modules in the system. To verify this, a series of line-to-sidestack simulations were conducted (results in table 4.4) varying the initial amount of modules from 4 to 10. Figure 4.30 show the 4-module case, and figures 4.28 & 4.28 show the 6-module case. As the number of modules is varied, the number of states that needs to be expanded in the search (which is proportional

[33]approx. step 0, 36, 76, 116, 140. Figure 4.30 shows successive figures of each new iteration for the 4-module line-to-sidestack reconfiguration

[34]this is because all bipartite modules in a sidestack have the ability to designate either of their parts as the 'base' or 'end' part of the helper module. Each base part choice will always keep the robot globally connected

Table 4.4: Reconfigurations: Line to Sidestack

Modules	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Execution Time (s)	23.03	38.34	57.76	81.56	105.6	137.9	167.1
Memory Required (MB)	7.79	9.99	12.28	14.39	16.70	19.16	21.43
SuperStates (Max)	52,177	70,211	88,008	104,213	121,991	140,537	157,849
SuperStates (Total)	102,084	172,412	263,343	370,347	496,490	645,600	808,273
Actions (connections)	16	26	39	55	74	96	121
Actions (unaided motion)	24	48	76	112	152	200	252
Actions (helper motion)	26	34	46	58	72	84	98
Actions (total)	66	108	161	225	298	380	471
Max Search Depth	26	33	39	45	51	57	63
Max Helpers	6	8	10	12	14	16	18

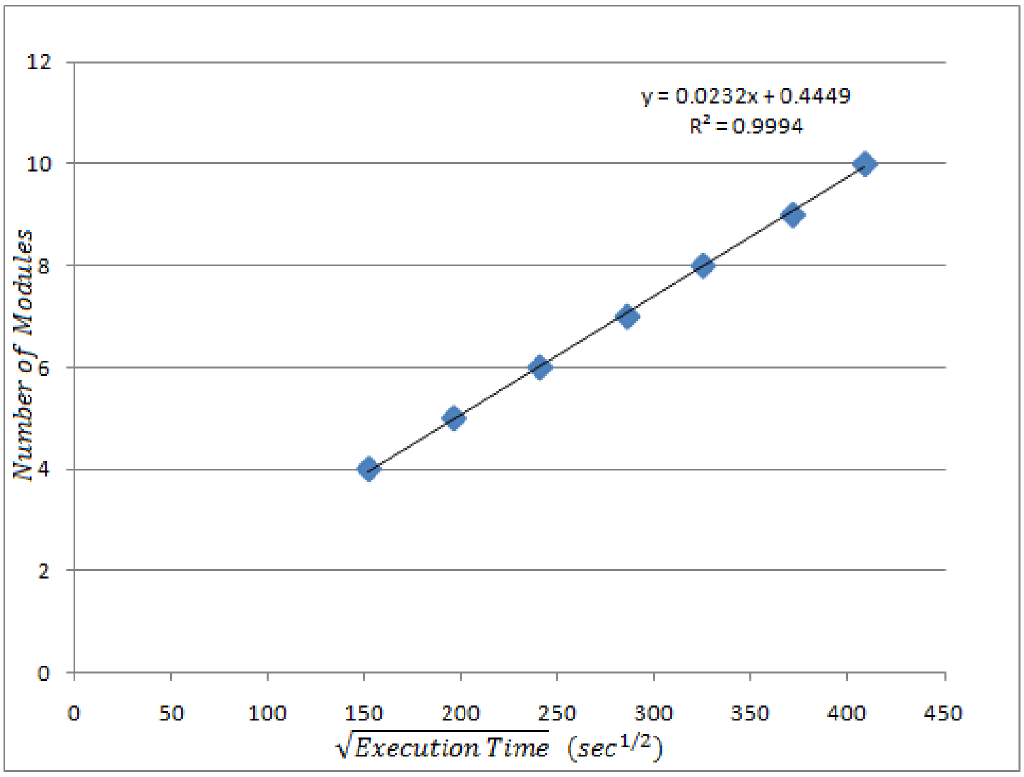


Figure 4.31: Sequence of line-to-sidestack reconfigurations showing CSRP's quadratic relationship between execution time and number of modules present

to execution time) grows quadratically. This is due to the serial nature of the CSRP; the addition of each module both enlarges the state space and is also one more module to relocate. The combination of these two factors forms the quadratic relationship of states needing to be searched. To see this, figure 4.31 shows how the number of modules for each sidestack experiment is linearly related (with a high R^2 value) to the *square root* of their execution times.

4.7 Discussion

The CSRP has shown to be able to reconfigure a 3R-based SRR between arbitrary shapes, though not all. It directs the serial motion of modules which does not make for a scalable SRP though it does simplify planning. Without the added complexity of parallelization of module movements and decentralized computing (like the DSRP and most SRPs reviewed) other aspects of planning could be focussed on. These aspects include the state representations of a module, the construction of state networks and the navigation functions used throughout them, and the coordination between a mobile module and a helper module. The success of these planning elements in isolation provides a firm prototype of the decentralized DSRP discussed next chapter. Nevertheless there still remain areas of the CSRP which can be improved on.

Completeness remains a difficult challenge, and while the CSRP has not yet achieved it, neither has any other SRP. For instance, chapter 3 SRP survey showed the Fracta planner could get stuck in ‘reconfigurable stalemates’. The MTRAN planner restricted itself to metamodules and thus is restricted to a subset of possible reconfigurations. The Claytronics planner can fail in reconfiguring between highly concave shapes, and Million Module

March can fail in forming shapes with internal cavities. Graph Signature is possibly complete given infinite time due to its method of simulated annealing which could eventually explore every reconfigurable option, but in practice is not correct either.

Proving completeness still remains a difficult, if not impossible task. In the author's view; proving completeness is tightly coupled with the challenge of developing a complete planner itself, and so the development of the first complete SRP will most probably precede the discovery of such a proof. It is hoped that future work on the CSRP will result in a general and complete planner, even if this is not provable. To be general and complete, a planner may be able to reconfigure between shapes with certain types of module but not others, owing to module design and not planner incapability. Such a property could be called 'mechanism completeness', whereby the planner will find a reconfigurable solution if one exists for such specific module the robot is composed of.

Reversibility is a desirable characteristic for planners and remains a challenge for the CSRP to achieve. Without regard to robot application, the property of reversibility is characteristic, though not necessarily indicative of; a planner choosing optimal reconfiguration paths because of the consistency of its decision making (this assumes all module actions are physically reversible). To fix this, Global Search's method of selected the 'next best suited' mobile module to relocate needs to be changed from being based on a absolute coordinate system as well as the robot's current configuration to being based on exclusively the robot's current configuration only. In this way the CSRP will use the same sequence of actions to reconfigure from one shape to another and then back again because each time Global Search will make the same decisions for the same reasons it did before. The only exception would be for symmetric structures where the planner must make a choices between two equally desirable options, however even if it took the ‘wrong’ option the reconfiguration is till guaranteed to proceed without breaking as it would only mirror its previous reconfiguration steps (by virtue of the structure’s symmetry). The property of reversibility can also be used to disprove a planner’s completeness property; if all module actions are physically reversible and a planner is not *reversible* it is therefore neither *complete*.

The correctness of the CSRP is a promising property. Not every SRP in the literature has this property, which this thesis defines as; a sequence of actions which successfully transform an SRR into the given goal shape. The Fracta planner for instance could accidentally reconfigure into shapes than the intended goal configuration due to the method of encoding configurations which did not guarantee uniqueness to the encoded configuration. The Claytronics planner also suffered some global disconnections during experimentations. The CSRP on the other hand is correct; it always predicts disastrous events such as global disconnections and potential collisions.

Chapter 5

Decentralized Planning for 3R-Type Modules

This chapter discusses the Decentralized Self Reconfiguring Planner (DSRP) of this thesis, its design elements and experimentations. This planner incorporates the same design structure as the centralized version (CSRP) described in the preceding chapter, though its implementation is different in order to break up the algorithm for distributed programming. Nevertheless it is recommended that the reader have read the previous chapter on the CSRP design beforehand, as routines such as *Transition Model* (sec. 4.5.6 - 4.5.8) have not changed and so not re-discussed here. Testing of the DSRP is with a Java simulator, the next step is to use the 'hardware in the loop' simulator discussed in the future work after translating into *C++*. Message passing is a major consideration in the decentralized planner as there can be significant time delays between module-to-module communications, and so even though communication is vital it needs to be kept to a minimum as much as possible.

Table 5.1: Comparison of CSRP and DSRP

Planner	**CSRP**	**DSRP**
Module Type	Superbot & MTRAN	Superbot & MTRAN
Lattice	cubic (3D)	cubic (3D)
Max. modules helper aids	1	0
Process	deterministic	deterministic[0]
Module Composition	homogeneous	homogeneous
Computing Architecture	centralized	decentralized
Module Movements	serial	parallel
Metamodules?	modules	modules
Simultaneous Actuations?	simultaneous	not simultaneous
Time Complexity / module	quadratic	sub-linear
Space Complexity / module	sub-linear	linear

The DSRP is a much improved design over the CSRP (table 5.1). The DSRP is not only decentralized, which is the natural computing architecture of modular robots, it can also plan for parallel motions of modules. This is important for the fast reconfigurations of robots and also for scalability; enabling large SRRs to reconfigure in reasonable time periods. Time complexity is sub-linearly proportional to the number of modules[1] which is a vast improvement over the CSRP. Though the DSRP is not yet finalized (helper classes are yet to be re-implemented from the CSRP), all significant challenges faced by decentralizing (including message passing) and parallelizing the CSRP have been accomplished.

[0]the DSRP is technically *framework stochastic*; asynchronous threading results in different reconfigurations for the same initial condition if run several times, but it is not *algorithmically stochastic* because it does not depend on any random number generator to operate like Claytronics, Graph Signature etc. do

[1]It is in fact of order $O(d + C)$ where d is the diameter of the module cluster and C is a large constant. This is sub-linear in all cases except for those involving a single file line configuration, which is the only scenario the DSRP executes in linear time

5.1 Algorithm

The DSRP runs as a single process but with multiple threads, one thread for each module. This changes the implementation of the *Local Search* routine discussed in section 4.5.5. This is because the CSRP relies on a single centralized lookup table that describes the global state of the robot. The principal reason for decentralization it to share the workload and increase robot robustness such that no one module is *special* or critical to the robot's operation. Therefore no module can be solely responsible of storing globalized information of the robot, as it forgoes these reasons behind decentralization and additionally such a module would ultimately become a bottleneck for the entire performance of the system. Instead all globalized information that exists in the CSRP needs to be distributed across the memory banks of all modules in the system. Additionally it is preferred that information always be shared in such a way that if any module malfunctions or breaks, updating any such data structure requires little message passing and thus not time expensive.

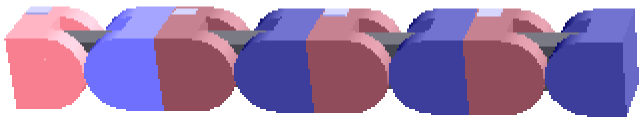
5.1.1 Local Search

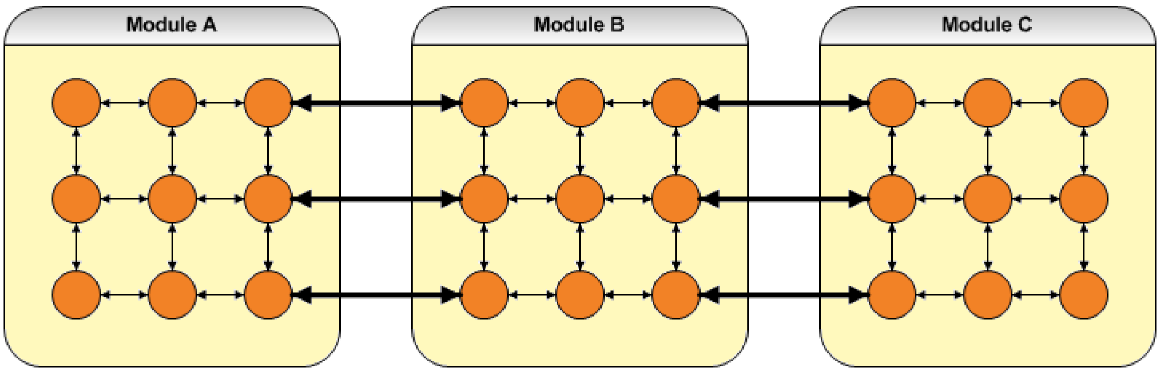
Local Search requires the building of a globalized state-network to support the use of a navigation function to direct the motion of mobilized modules (sec. 4.5.5). In DSRP implementation each structural module is responsible for computing and storing a portion of the total state network in their own memory, called their *Local State Network* (LSN). The LSN of a structural module is defined by all states that require physical connection to that structural module. Because this is a set number (up to 648 states for static modules and up to 9510 states for helper modules) per module, memory requirements grow linearly with every module added, but as each module also *supplies* more memory when added the robot's size will never be limited by memory constraints and a global state network can be build to an unlimited size. As additional structural modules are added to the robot system, each will take responsibility for representing the states in their own vicinity, i.e. the states added to the global state network because of their presence.

Each structural modules independently builds its own LSN in the same way outlined in the Local Search algorithm 4.4 of section 4.5.5, however are restricted to the set of states that require physical connection to the structural module. The LSN is first populated with all states that a mobile module can possible exist in around the structural module and then linked up according to which actions are valid at any particular state using the *Transition Model* (sec. 4.5.6 - 4.5.7). Transition Model's implementation does not change for the DSRP because none of its functions require information of the global state, only the local state and actions of modules. Thus every module can store their own copy of Transition Model (it is only 93.8kB), because it does not change once compiled for a certain module type.

Once all structural modules have completed building their own LSNs (all computed in parallel, thus an efficient constant-time exercise), the last task before the robot has a globalized state network is to link the local state networks together forming a single continuous global network. This is needed for a navigation function to operate on, assigning values of desirability to different states directing the motion of mobile modules. In the DSRP case this process is much faster[2] owing to the large degree of parallelization; each module's processor updates the values of its own states. To link LSNs together, the state-action pairs that transition to states that belong to another module's local state network (called *cousin* states) must be identified. This is relatively easy; when a module is linking up its own LSN

[2]depending on the configuration of the robot. Execution time can be at most linear with the number of modules as it always is for CSRP, if the robot in the DSRP case is configured in a single file line shape, however for every other configuration it will have sub-linear execution time


(a) Three structural modules connected in a line configuration and one mobile module; the lighter colored leftmost module



(b) State network

Figure 5.1: Visual aid of the decentralized organization of the network or states

(after it has populated it with possible states) can checks that taking a particular action from a particular state does results in a state that does not exist in its own LSN, it can safely assume the state must exist in at least one other module's LSN[3]. However a communications process between modules is required to determine *which* other module the transitional state belongs to. When this is found; the LSN of either module link via the action-link that exists between these two states.

A simplified visual of the structure of linked LSN's is shown in figure 5.1, which shows a state network (Fig. 5.1b) based on the robot configuration of figure 5.1a. Each LSN is represented by a collection of states (orange circles) belonging to a particular module

[3]This assumption is always valid on the condition that all other modules have finished populating their own LSNs, as a Superbot module can never take an action (a complete list is located table 4.1) that would result in a state not of the global state network because there exists no such action that disconnects both module parts. So if the module is always connected to at least one structural module's connector then it must exist in at least that structural module's LSN

(yellow background). States within a LSN are interlinked (thin black lines), and LSNs are linked to each other in some instances (thick black arrows). These links are the actions a module could perform in any one state. There is often multiple actions a module could take in any one state such as moving left, right, up or down the action-arrows of any state (circle) in Fig. 5.1b. The leftmost module in Fig. 5.1a does not have a LSN because it is has been mobilized, it will be taking advantage of the LSNs structural modules have computed to make decisions about how to move.

To build these inter-module links (thick arrows), a module is programmed according to algorithm 5.1 which initiates inter-module communications if a transitional state does not exist in a module's own LSN. It may know the state exists somewhere, but not until it communicates with surrounding modules will it know which module it exists in and thus which LSN is links to. As the Transition function (sec. 4.5.6) will return a full description of the state in question, the module will know where the state will be located[4] and so it can send a message to a connected structural module closet to that direction which would be most likely to have that state. If it doesn't it can relay the message to module's further out, capped by a maximum search depth. Each module that receives a message concerning the state will evaluate to message and return an 'affirmative' message including its identification number. In fact up to three modules can send back such reply because they states can overlap the domains of structural modules in cases of *concavities* discussed later in section 5.1.1. Upon return of a reply message, inter-module action links are noted in both modules because all 3R-module actions are reversible, so validating a transition in one direction also validates it in the other. The linking of LSNs is time expensive (due to multiple message passing between modules), though once built can be re-used indefinitely until a

[4]in fact it will be located exactly where it was before the action; as only connector-actions can change transition a state from one module to another and connector-actions never move modules

local change occurs to one of these structural modules, such as it mobilizing.

Algorithm 5.1 Decentralized state network constructor

```
1: populate local states of Module
2: for each State/Action pair do
3:     (State) NeighborState = TransitionModel(State, Action)
4:     if NeighborState does not belong to Module then
5:         MESSAGE(to a connected module in NeighborState's direction): "does NeighborState belong to your local set?"
6:         for each reply of 'yes' by AnotherModule do
7:             record NeighborState as valid state on a different module; AnotherModule
8:         end for
9:         if no replies then
10:            FAIL: "a valid state does not exist anywhere yet. This means other modules have not yet finished completing their own LSNs"
11:        end if
12:    end if
13: end for
```

However a problem with that arises in multiple reply messages is the nature of the network changes from the CSRP case of only one-to-one links exiting to many-to-many links. This is because the same physical state can exist in the domain of multiple structural modules, so a mobile module must choose which state representation under which module's domain it should transition to. This is discussed in sec. 5.1.1.

When the global state-network is finally constructed, a navigation function borrowed from dynamic programming (sec. 3.2.3) is able to spread through it designating values of desirability to each state for mobile modules to consider. Global Search selects a goal state for one or multiple mobile modules to move towards, the module that houses the goal state (the 'goal module') in its LSN will propagate the navigation function outwards just as CSRP does for the entire state space, outlined in algorithm 4.4 in section 4.5.5. The DSRP's

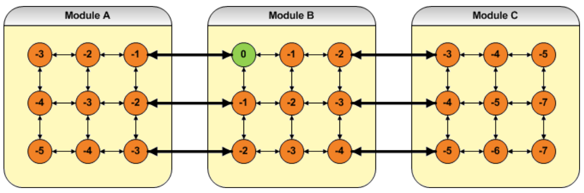
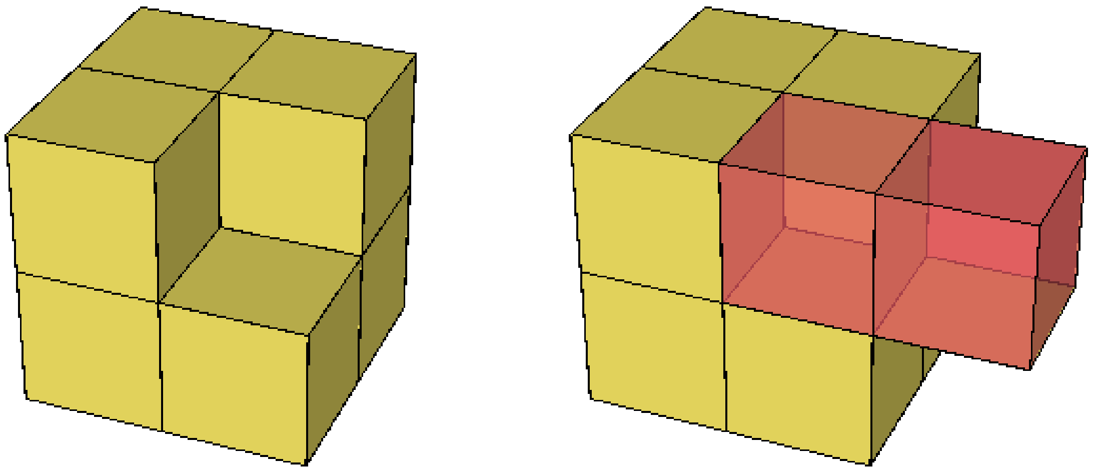


Figure 5.2: Navigation function spread throughout a DSRP state network

navigation propagates beyond the boundaries of the ‘goal module’ whenever any state in its LSN is updated (algorithm 4.4 line 14) by also notifying the cousin module (if it has one - it will know if it does have one because of the presence of an inter-module link) of the update. It does this as part of the normal expanding of state-nodes, examining the children nodes to determine if they require updating also. In the DSRP case it just so happens one of these nodes can be a cousin node (a linked state on a different module). Module’s send these messages in the form of offering what could be a potential value of the cousin state in a cousin module. If such a value is better than what that cousin state’s value currently is, it will update, and the navigation function will spread throughout that new module as well. The end results in a propagation of the navigation function that resembles the final outcome of the CSRP navigation spread as seen in Fig. 5.2.

Concavities

Concavities are lattice-cells over a robot’s topological surface that are adjacent to two or more cells occupied by structural modules. Concavities can exist over the surface of a robot as exampled by the ‘missing’ cube in figure 5.3a. They lead to the undesirable effect of states being registered in the local space of more than one structural module, such as the red bipartite module position in Fig. 5.3b. In this case the connector arrangement of a Superbot module is such that a state in this red positioning of a module could be connected



(a) A concavity cell which is adjacent to 3 structural module parts

(b) A mobile module (red) with one part located in the cavity

Figure 5.3: Concavity

up to 2 of the structural modules of Fig. 5.3a colored yellow. As inter-module communication is restricted, both structural modules will not be aware they share a common state with each other and will both unnecessarily double up work by computing the same state parameters required by an instance of the *State* class. The benefit of a centralized state network is all states are stored in one location so attempts to re-instantiate the same state area easily recognized by checking the single state network to see if it exists yet. This is not a massive computational inefficiency however, the proportion of doubled-up states to total states present over the shape of Fig. 5.3a is only 2.21%[5].

[5]Proof: As mentioned in sec. 4.3 there are 108 states for every connectable face assuming no other modules exist on the other side of the plane made by the connecting face. All faces that are not part of the concavity, of which there are $4+4+4+3+3+3=21$, conform to this. For each of the 3 faces within the concavity, each is able to support one module position that is aligns parallel with the face (12 states) and two positionings in each of the other Cartesian direction (24 states each) that align the module perpendicularly to the connectable face. $21 \times 108 + 3 \times (12 + 24 + 24) = 2448$ states in the system total. The number of doubly connected states; 3 out of the 6 possible part-states of the part inside the concavity in Fig. 5.3a will be double connected. This red positioning could be aligned in 3 directions to fill this cavity (X, Y or Z) and each time, there are 6 possible part-states the non-connected part can be in for each of these positionings. Therefore there are $3 \times 3 \times 6 = 54$ doubly connected states possible. The proportion is $54/2448 \approx 2.21\%$

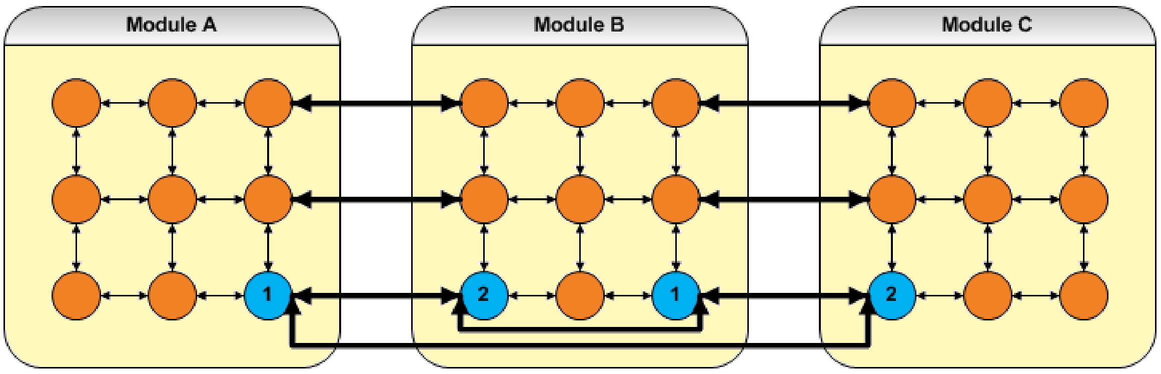
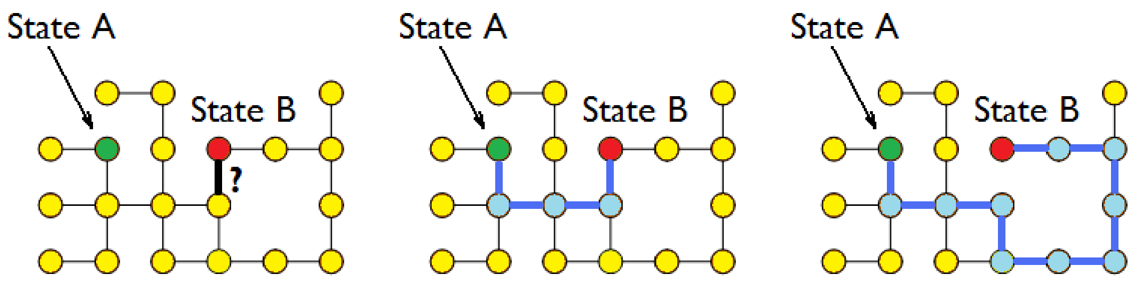


Figure 5.4: Visual aid of the decentralized state network that has a concavity. Duplicated states are colored blue and identified by numbers. Shown are two instances of state-1 and two instances of state-2

The real complexity concavities add to the creation of a state network is changing the one-to-one relationship state/actions pairs have with other states to many-to-many exampled by Fig. 5.4. Because concavities allow certain states to belong to more than one structural module's LSN, state/action pairs can link to several states recorded in different modules and vice versa. One option is to enforce a one-to-one linkage by only accepting one of the reply messages in algorithm 5.1. However it is not obvious to distinguish which module should be chosen from potentially three replies. If the choice is random, such as accepting whichever module was the first to reply and then ignoring all subsequent replies, then the state network can become fragmented which would eliminate the navigation function's guarantee that all (directed) transition paths for mobile modules are optimal.

Fragmented means that some valid action-links have been ignored, links throughout the state network can become unnecessarily sparse, and what may have been an optimal reconfiguration path using some of these ignored action links is now rerouted by the planner that is not aware they actually constitute valid actions. As a simple example consider figure 5.5. If a navigation function attempts to direct a mobile module from state-A to state-B,



(a) State network, with a potentially ignorable action linkage near state B (in bold)

(b) If the bold action link is not ignored; modules find the truly optimal path of 4 atomic actions from state-A to state-B

(c) If the bold action link is ignored; modules find the seemingly optimal path of 10 atomic actions from state-A to state-B

Figure 5.5: Example of state network fragmentation. Shown is a global state network (module domains not shown)

it could choose the optimal path in Fig. 5.5b, or the seemingly optimal path in Fig. 5.5c depending on the bold link in Fig. 5.5a is ignored or not.

A visual example of a state network around a concavity would resemble something like figure 5.4. This is similar to the example seen in Fig. 5.1b except there are two additional inter-module links due to overlapping states. The overlapping states (blue) are numbered signifying where the same state exists on different modules (state-1 exists on module A and B, state-2 exists on module B and C). Notice the complexity arises in the multiple paths a planner can take; if the planner search is currently considering state-2 on module C and takes the 'left' action it could either go to state-1 on module B or state-1 on module A. In real terms this translates to does the search progress by sending a message to module B or C? This DSRP sends messages to both. Although this incurs some duplication of work it ensures concurrency between the state stored in each module (the values of the state stored on each module always agree) preventing fragmentation of the global state network. Happily, the intra-module links are always one-to-one and it is only the inter-module links which can be many-to-many due to concavities. This means their occurrences in the state network are uncommon, because inter-module links are uncommon due to the fact that only connection-action can initiate them (there are much more movement-actions than connection-actions as seen in table 4.1) and only when in the direct proximity of another structural module. The example seen before with a low 2.21% duplication of work also agrees with this. And even though the duplication of work could have been higher if the example robot structure in Fig. 5.3a had more concavities, there comes a limit as to how many concavities can exist in a set region, so duplication of work is at least always bounded by an upper threshold.

An alternative option for the DSRP could have been to scrap what caused the problem of duplicated states in the first place; the way in which how the position of a module is defined in the state representation as developed in the CSRP (sec. 4.3). The position of a module state is defined as being the location of the connected part[6] of the two-part module Superbot. The advantage of this (discussed in sec. 4.2.1), as opposed to being defined by which connector the connected part is connected to, is in the cases of concavities; what would have been multiple states which are all functionally identical (if positions are defined by connectors), is instead a single state (the planner assumes a module in that state will connect to all connectors it's adjacent to, not just one). This compresses the searchable space that needs to be searched making the CSRP a more efficient planner. Ironically for the DSRP case, the efficiency gain this produces is matched by the 'duplicated computation' (previous paragraph) it also causes. However by changing the state position to being defined by the connector is connects to (as Million Module March does; sec. 3.7), and states are only considered to be connected to one other connector at a time (such that a module doesn't have *two* positions values under this new definition), then such a state is unique

[6]more specifically: the geometric center of the connected part

to that connector. The connector in turn belongs to the module it is a part of, and so that structural module will have a set of states that are unique to it. For a state in a cavity; its state representation must specify which connector it is connected to, and such a state will belong to that connector's module. Given the state can possible connect to several structural modules in that *location*, no complexity occurs because each connection represents distinct states.

A major disadvantage of the alternative option above is that is also foregoes the reduction of a state network's dimensionality[7] discussed in section 4.2.1. This stated that the *location* definition over the *connector* definition gave rise to less connection-actions needed in Transition Model's repertoire. This reduction in dimensionality reduces search time considerably and for this reason the *location* definition is kept along with the duplication of states that requires symmetric treatment to be kept concurrent (such as sending messages to both module B or C discussed in the example before).

5.1.2 Message Passing

Communication between modules is frequently a limiting factor of planner performance in any modular robots dues to significant time delays and low bandwidths. The *hardware in the loop* used to test DSRP implementation is no exception. Inter-module communication for the DSRP is via hardwired message passing. If a message needs to be sent between modules not directly connected, it is relayed by interim modules. Messages are packeted into 80 bytes blocks. To take advantage of this knowledge, wherever possible; messages larger than 80 bytes are compresses into 80 bytes, and messages smaller than 80 bytes are bundled with other small messages and sent as one. This is because the smallest message

[7]the maximum number of actions to choose from any one state-node

that can be sent is 80 bytes, and considering it is expensive to do so, it should contain the maximum amount of information possible.

Table 5.2: DSRP Message Types

Code	**Name**	**Description**
0	*CousinState*	A module asking other modules it a cousin-state exists in their LSNs
1	*CousinStateReply*	A reply to *CousinState* message (only if state exists in LSN)
2	*QValueOffer*	Used when a state's value changes, the state notifies its cousin state
3	*QValueEnquiry*	A mobile module enquiring about the value of its cousin state
4	*QValueEnquiryReply*	A reply to *QValueEnquiry* message (always replies)

The DSRP supports five types of message listed in table 5.2. Message types 0 and 1 are used to link the LSNs of different modules in the construction of a global state network as part of Local Search's routine. Message type 2 is used as part of the dynamic programming implementation to propagation a navigation function around the decentralized state network. If a state updates on one module, it is able to notify a cousin state of this update using message 2. When a state network is both constructed and a navigation function has spread through it, then messages 3 and 4 can be used by module modules to enquire adjacent states as to their values. It uses these messages to check the values of each neighboring state and picks the state with the highest value to next transition to. Each module stores received messages in a circular buffer, and reads messages on a FIFO (First In First Out) basis. Overflowing the buffer causes modules only throw warnings at this stage, but do not terminate module operations as the messages will eventually get re-sent.

The DSRP design attempts to minimize all instances of message passing. One way to do this is by compressing information to fit inside a single packet using hash codes. The advantage of modular robots is that each can be programmed beforehand with the entire DSRP planner including all hash code encoding/decoding routines, and so handshaking is required on the DSRP level, the format of messages is always as expected. Encoding a state is required when one module asks another module if a cousin-state exists in its LSN. An example of encoding a state (algorithm B.1) is included in appendix B. It allows the DSRP to take advantage of the fact that a 3R module can only assume 216 different states per position and that this number is less than 255; the highest value of an unsigned byte. So by hash encoding, all information included in a state except position is uniquely represented by a single number 0 to 215, compressed into a single byte. If the robot is less than 128 modules, each Cartesian direction compresses into a single byte also, totaled 4 bytes per state. A receiving module can decode a message to recreate original objects by a reversal of this process. Algorithm B.2 shows the process of decoding an encoded state object.

5.1.3 Parallelization

Parallelization is the simultaneous movements of multiple surface-moving modules, which the DSRP supports. To do this three planning components need inclusion/updating:

1. **Global Search:** Something similar to Million Module March (MMM) (sec. 3.7) would be suitable, whereby certain modules selected have the ability to lock down immediately surrounding modules to initiate movement without ‘traffic jam’ conflicts. These conflicts occur when two modules side by side are each able to become mobile without violating global connectivity, but not both. In this instance Global

Search needs to make a decisions about which should move, which MMM incorporates with one module locking the other (whichever was fast enough to send the locking message first). So far the Global Search has not been implemented dynamically, only hard-coded, so this is a matter for future work

2. **Universal Navigation Function:** The navigation function modules use to make movement decisions needs to be universally accessible and applicable to all mobile modules. As is, the CSRP implementation is not far off but the representation of the goal needs to change from a single state somewhere to a *region* that can accommodate multiple modules. This can also warrant the continual updating of navigation function values as some modules inevitably block the paths of others necessitating the updates in order for modules to reroute. So far in this implementation, modules will simply wait for modules in front to get out of the way

3. **Collision Avoidance:** A dynamic collision avoidance function needs to prevent surface-moving modules from colliding into each other. This is a dynamic process that prevents some modules from accessing certain parts of the state space at certain periods of time in contrast to the existing collision avoidance function which assumes the every other module is always static for one surface-moving module's locomotion. This implementation is achieved also with the use of locks, whereby a module can reserve certain sections of 3D space (lattice cells) for its exclusive use to move through. At the conclusion of every successful atomic action a module takes, it releases its locks over the space it moved through to allow other modules to pass through as well. As this is a decentralized process, modules have to ask neighboring modules if they have locks over a certain regions before moving

5.2 Evaluation

This section evaluates the DSRP in terms of its performance and capabilities. It begins with an analysis outlining performance properties and continues with examples of DSRP-directed reconfigurations.

5.2.1 Analysis

Memory Usage

DSRP memory usage is linearly proportional with the amount of modules in a system. This is because each module only needs to store a set amount of information; its LSN (Local State Network) a set amount of messages, which are independent[8] of the number of modules in a cluster. This is a distinct advantage of the CSRP, as the DSRP-led robot will never be limited in size due to exceeding memory capacity. As a modular robot grows in size, memory requirements increase for both the DSRP and CSRP, however the DSRP is the only planner who's available memory capacity grows with it. As mentioned in sec. 4.6.1; the maximum amount of states a module can introduce to the global state network is 19020, requiring 4.71MB. Thus each module's in the DSRP needs to be able to reserve 4.71MB in its own memory to store its respective LSN. Messages are confined to 80 byte packets, and most modules do not store beyond 20 messages before they have a chance to process them. In addition each module needs a complete copy of the DSRP's source code totaling 170kB. Thus 5MB from each module would be adequate to store and run the DSRP. This is not much, as mentioned in section 4.6.1; off-the-shelf SD cards are well suited for module installation and range in the GB's.

[8]they are only locally dependent on the placement of modules about them

Execution Time

The execution time of the DSRP is much quicker on average that the CSRP. For all reconfigurations except those involving single-file line configurations; the DSRP takes sub-linear time (with the number of modules present) to solve reconfigurations.

Execution time complexity is: $O(d + C)$ where d is the diameter of the module cluster and C is a large constant representing the time taken for each module to independently set up their own LSN. This is a constant as LSNs can only be so large (19020 states maximum if it is a double-helper[9]), and therefore upper-bounded. For a small number of modules the constant C is the dominant factor, however for a large number of modules d becomes the dominant factor of execution time. Thus best case reconfigurations include spherical shapes which have the best d to number-of-modules ratio. Here, $O(d) = O(\sqrt[3]{\frac{6.n.v_{module}}{\pi}}) = O(\sqrt[3]{n})$ where n is the total number of modules and v_{module} is the volume of each module. This is very much a sub-linear relationship. The other extreme, when all modules are lined out end to end in a straight line, then $O(d) = O(n)$, and execution time is linearly proportional to the number of modules present.

Correctness & Completeness

Like CSRP, the DSRP is *correct* but not *complete*. It will not proceed with actions that result in global disconnections, and modules are prevented from colliding with their ability to *lock* down regions of 3D space they will move through, prohibiting other mobile modules from entering as they move in parallel. The DSRP is not complete, in fact due to time limitations the object class defining helper module movements cannot been ported and

[9]double helpers are those which can act as a helper from either side, i.e. each part is allowed to detach and be the end part. As helpers contribute 9510 states each to the global state space, a double helper will contribute $2 \times 9510 = 19020$

re-implemented from the CSRP yet so the DSRP's reconfiguration options are limited. Nevertheless it successfully shows that it is a faster and all round more efficient algorithm than the CSRP in section 5.2.2. As mentioned in the previous chapter completeness is a difficult if not impossible property to prove, nevertheless it is hoped that future work with the DSRP will result in a planner that is complete.

5.2.2 Examples

This section demonstrates and compares DSRP-directed reconfigurations. Without the use of helper modules, testing the DSRP has been confined to examining the locomotion of mobile modules in ways that do not require coordinated motion. Fig. 5.6 shows one experiment involving 4 mobile modules (lighter colored modules in Fig. 5.6a) moving from an initial layout down a single file of static modules in parallel to bunch up at the opposite end. This kind of simulation demonstrates performance gains of the DSRP over the CSRP, allowing multiple mobile modules to share usage of global state networks instead of rebuilding them after every module relocation. A comparison is then given between the reconfiguration of Fig. 5.6 for both DSRP and CSRP planners in table 5.5, as well as a look into the message passing required for the DSRP case in table 5.4. The translation of the DSRP into C++ and the porting of it onto the hardware in the loop simulator has not been completed, and so the DSRP has only been simulated at this point on a centralized computer in Java. This is simulated with the use of a scheduler and a module class, cycling between threads located in each module.

The simulation of Fig. 5.6 is replicated with varying numbers of mobile modules and static modules with respective performance parameters listed in table 5.3. Fig. 5.6 represents experiment 'D7' which has 4 mobile modules moving along a line of 6 static modules

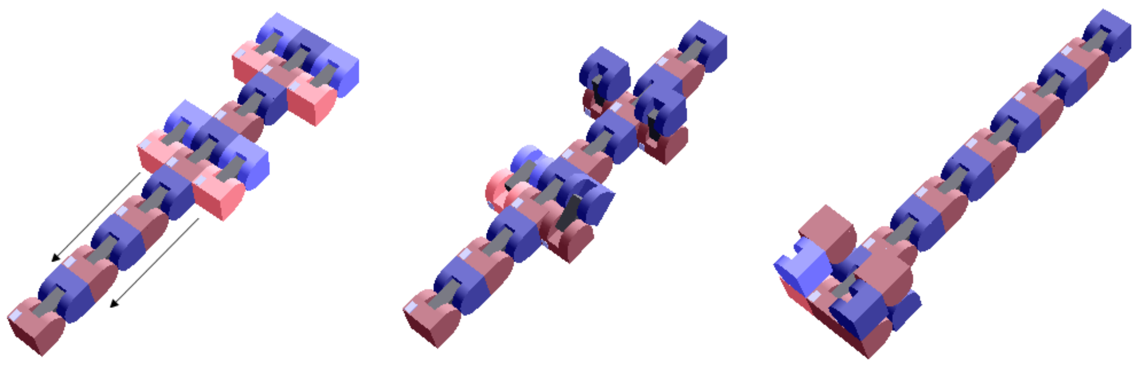

(a) Initial configuration (b) Module modules in motion (c) Final configuration

Figure 5.6: Experiment D7

and the other experiments are conducted similarly. Performance parameters are;

Simulation Time (s): Total time to conduct the DSRP simulation

Avg. CPU Util. (s): (Average CPU Utilization) The average time spent in each module's thread

Avg. Memory (kB): (Average Memory) The average amount of memory each module used

States (total): The number of states in the global state network

Actions (connection): Total connection-type actions from all modules

Actions (motion): Total unaided motion-type actions from all modules

Actions (total): Total of all actions for all modules

Messages: Total messages count between all modules

The key result here is that as the number of mobile modules increases for any fixed number of static modules, the average CPU utilization of modules does not increase significantly. The reason it increases at all, is because mobile modules usually compute more than static modules (as seen next in table 5.4) and thus bump up this average. As seen

Table 5.3: DSRP Experiments

Experiment	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
Mobile Modules	1	1	1	2	2	2
Static Modules	18	30	42	18	30	42
Simulation Time (s)	2.63	4.62	4.98	4.37	6.90	8.38
Avg. CPU Util. (s)	0.138	0.149	0.116	0.219	0.216	0.190
Avg. Memory (kB)	289.2	292.2	293.5	274.7	283.1	286.9
States (total)	7,992	13,176	18,360	7,992	13,176	18,360
Actions (connection)	35	59	83	70	118	166
Actions (motion)	73	121	169	146	242	338
Actions (total)	108	180	252	216	360	504
Messages	4,211	8,915	15,044	4,942	10,138	16,759

Experiment	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**	**D13**
Mobile Modules	4	4	4	4	6	6	6
Static Modules	6	18	30	42	18	30	42
Simulation Time (s)	2.08	7.39	10.24	15.52	10.18	15.14	21.24
Avg. CPU Util. (s)	0.208	0.336	0.301	0.337	0.424	0.421	0.443
Avg. Memory (kB)	193.1	249.8	266.4	274.4	228.9	251.6	263.0
States (total)	2,808	7,992	13,176	18,360	7,992	13,176	18,360
Actions (connection)	34	130	226	322	178	322	468
Actions (motion)	84	276	468	660	390	678	987
Actions (total)	118	406	694	982	568	1,000	1,455
Messages	1,966	6,214	12,265	19,762	8,092	14,108	22,952

the average memory use is also fairly constant. The average rises slightly as the ratio of static modules to mobile modules increases because mobile modules are not required to compute a LSN (they use the LSNs of static module by enquiry), they only need to store received messages. The total messages transmitted and received is contributed from both static modules in the construction of the global state network and from mobile modules navigating it. Both these contributions are linear with the number of modules, resulting in a linear total. A similar relationship holds for the global state space size as well. As these experiments operate on a line shaped configuration, this represents the worst case scenario for the DSRP, resulting in these linear (not sub-linear) relationships.

Table 5.4 offers a closer examination of the message passing of a typical experiment (D7) needed to support a DSRP reconfiguration. Module classes 'M' and 'S' signify *mobile* and *static*, and the five message types either transmitted or received are detailed in table 5.2, section 5.1.2). One clear distinction between both modules types is the CPU utilization of each. The average mobile module required 10 times the utilization than static modules did. This is because once static modules have set up the global state network their job is practically over; they only need to respond to mobile module queries. The mobile modules on the other hand are continually determining their next choice of action, waiting on message replies and preventing collisions. To prevent collisions the *Collision Detection* routine (sec. 4.5.7) is called by the mobile module to compute the encroaching lattice-cells. It then checks these cells with nearby modules to check if they are unlocked before proceeding incurring further computation, all of which is time expensive.

The main form of messages are the *CousinState*/*CousinStateReply* types which modules use to continually discover the presence of others in their local proximity. This is needed both in the creation of a global state network and also avoid potential collisions. Static

Table 5.4: Message Statistics from Experiment D7

Module ID	0	1	2	3	4	5	6	7	8	9
Module Class	M	M	M	M	S	S	S	S	S	S
CPU Utilization (ms)	503	420	468	419	56	20	51	20	98	30
Actions										
Connections	10	10	7	7	0	0	0	0	0	0
Unaided motion	25	25	17	17	0	0	0	0	0	0
total	**35**	**35**	**24**	**24**	**0**	**0**	**0**	**0**	**0**	**0**
Messages Transmitted										
CousinState	31	36	34	34	1	1	1	1	1	1
CousinStateReply	64	86	83	70	22	42	52	64	66	45
QValueOffer	0	0	0	0	72	121	121	121	118	38
QValueEnquiry	11	11	7	7	0	0	0	0	0	0
QValueEnquiryReply	0	0	0	0	2	4	6	8	12	4
total	**106**	**133**	**124**	**111**	**97**	**168**	**180**	**194**	**197**	**88**
Messages Received										
CousinState	64	86	83	70	22	42	52	64	66	45
CousinStateReply	157	182	172	172	4	7	5	5	3	2
QValueOffer	0	0	0	0	49	121	121	121	110	69
QValueEnquiry	0	0	0	0	2	4	6	8	12	4
QValueEnquiryReply	11	11	7	7	0	0	0	0	0	0
total	**232**	**279**	**262**	**249**	**77**	**174**	**184**	**198**	**191**	**120**

modules only ever need to send the message once whilst they or their neighboring static modules are not mobilized because otherwise their local space does not change. This is with the exception of mobile modules moving along beside them however a static module does not need to be aware of their presence (besides when forming a connection) because their computational role is simple to build the global state network. This is turn in independent to where mobilized modules are currently located, they simple fill one of the many states the global state network plans for. Some message type are not transmitted or received by one of the module classes due to their nature. A *QValueOffer* type is only needed between static module in state network creations, and *QValueEnquiry* are only ever sent by mobile modules and only directly to static module to enquire the value of states within their LSN.

Table 5.5: DSRP and CSRP comparison

Experiment	**D3**		**D6**		**D13**	
Mobile Modules	*1*		*2*		*6*	
Static Modules	*42*		*42*		*42*	
Planner	**CSRP**	**DSRP**	**CSRP**	**DSRP**	**CSRP**	**DSRP**
Net CPU Use (s)	4.14	4.98	11.92	8.38	91.81	21.24
Net Memory Use (MB)	10.06	12.33	10.03	12.33	10.30	12.33
States (max)	14,989	18,360	14,934	18,360	15,344	18,360
States (total)	14,989	18,360	29,729	18,360	85,031	18,360
Actions (connection)	83	83	166	166	468	468
Actions (unaided motion)	169	169	338	338	987	987
Actions (helper motion)	0	0	0	0	0	0
Actions (total)	252	252	504	504	1,455	1,455

To compare DSRP performance against that of the CSRP, table 5.5 shows the results of running experiments D6, D10 and D13 with both planners. As the DSRP does not yet able to plan with helper modules, the helper class in the CSRP was disabled for fair comparison. The equivalent action counts between planners indicate both decide on the same course of actions for reconfigurations. This shows that the decentralized state network of DSRP is still able to produce optimal relocations of modules given CSRP does. In the ‘D3’ experiment of a single mobile module, the CSRP performs marginally better. This is because the CSRP is not forming a full state network like the DSRP is, it builds outwards according to a BFS search until is find the goal state it is looking for. At this point does not require further building as it will not be reused and the merits of the BFS search guarantee the first found solution is optimal. However in experiment ‘D6’ DSRP becomes the more efficient planner still constructed with 18,360 states. The CSRP on the other builds one network to move the first module in serial which then becomes a static module by the time the second module is mobilized. The state network it not reused because it has changed by virtue of the previously mobile module becoming a static. Both these networks are about 14,900 states each totaling in 29,729 states searched to reconfigure. By experiment ‘D13’ with six module modules, this increases significantly again to 85,031 resulting in a much longer net CPU time of 91.8 seconds compared to that of the DSRP’s 21.24 seconds. The ‘net CPU use’ is the net total of threads execution time on the DSRP, and is simply the single execution time in the CSRP’s case. For cases involving 2 or more modules moving in parallel the DSRP will make better use of the robot’s computational resources. Though what these simulations of the CSRP do not account for is the message passing needed for a CSRP to operate on one module, directing all other modules. This gross use of global message passing would slow down such an algorithms execution even though its simulated time is shown to be superior than the DSRP in experiment ‘D3’. Since most reconfigurations do involve multiple module relocations, DSRP will be a much faster algorithm than CSRP most of the time. The ‘net Memory use’ is the maximum sum total of memory required at any point by the planners in the building of state networks. The

CSRP always uses less memory than the DSRP case because of the partial building of any state network, the DSRP's memory requirements as discussed before is upper bounded. The DSRP only needs to reserve about 5MB on each module, and so the improvement of execution time from a quadratic to an 'at most linear' relationship with the module count significantly outweighs prospect of saving one or two megabytes on each module.

5.3 Discussion

The DSRP has shown to be a more suitable choice for a modular robot over the CSRP. Its efficient reuse of state networks is the key for a faster execution time, and enabling modules to move in parallel makes for faster robot reconfigurations. The CSRP serves as a good prototype useful in simulation to test reconfiguration strategies but eventually such a program needs to be installed onto a modular robot and in this case the DSRP is a natural implementation to the robot's distributed computer architecture. This not only takes advantage of being able to run multiple threads on different microprocessors but also does not require global broadcasting; messages are only transmitted between modules in close proximity, minimizing communications delays.

The mobilized modules have shown to bear increased computational burdens over static modules in a DSRP reconfiguration. This imbalance is inherent to the DSRP design so does not necessarily pose a problem to be solved but nevertheless could be accommodated for. Fully functional modular robots will need to share computational resources with various tasks other than planning, including stability analysis, sensing, localization etc. Computational resources will no doubt need to be managed, and the most efficient way to manage the DSRP will be to allocate the planning threads onboard mobilized module greater priority than those onboard static modules, especially when those static modules are not connected to any mobile modules. This would decrease delays for planning related messages and provide for more rapid computation of reconfiguration solutions.

The DSRP compares well with other SRPs reviewed (chap. 3). For one, like the CSRP it does not restrict SRR configurations to those of metamodules like the MTRAN planner (sec. 3.5), which although simplifies planning, also reduces SRR versatility. The DSRP is additionally deterministic (as is CSRP), enabling greater predictability than all SRPs reviewed except MTRAN. Its decentralized, parallel & non-metamodule design is rivaled only by Fracta (sec. 3.4) and its sub-linear order execution time is rivalled only by the Million Module March (sec. 3.7). It does not however plan for heterogeneous composition as Graph Signature does. The benefits of heterogeneous design (sec. 3.3.3) to SRR is a worthwhile consideration in a SRP though was never part of the DSRP design goal. Given that most of its routines are general (hardware independent), minimal re-implementation is required to instantiate it to operation on different module hardware. Conclusively the DSRP is a tractable, pragmatic SRP suitable for any Suerbot or MTRAN based SRR.

Chapter 6

Conclusions

6.1 Summary of Work Completed

This paper has taken a novel approach to planning reconfigurations of modular robots in a tractable way. A general understanding of previous self reconfiguration planners revealed that those which were purpose-built for particular module designs can offer more efficient reconfiguration strategies than generic planners that plan for metamodules. The downside is such planners are limited to the modules they were designed for whereas generic planners are portable. The contribution of this thesis has been towards generality; to break this trade off and draw advantages of both types to build an efficient yet portable Self Reconfiguration Planner (SRP).

Module designs are widely varied and survey papers have been published to collate and compare these designs against each other yet research shows SRPs are also widely varied though no such comparative survey yet exists. As part of the background research for this thesis, to draw from the present state-of-the-art in planner design, this paper includes a survey of some of the most influential and original SRP algorithms.

An understanding of module representation is investigated with emphasis on exploiting symmetries to reduce the search task as much as possible. This is extended to a cooperative module representation that simplifies (provides an abstraction for) the planning process of interdependent modules. The central design is based on hierarchical search to compute close-to-optimal reconfigurations for an SRR. It is a decentralized program that supports parallel motion of modules and executes in sub linear time with respect to the number of modules composing a robot. The design is deterministic, which is not common amongst SRPs due to the vastness of a robot's configuration space, however the merits of reliability and optimal module locomotion were deemed significant enough to be designed this way.

This is the first algorithm that plans general reconfigurations for surface moving modules whilst addressing module's physical characteristics and constraints (its native kinematics). Most of the design is hardware independent, requiring minimal effort to port to different module hardware. A fully generalized design is left for future work, and when completed; would require no hand-coding of motion primitives, and thus would require next-to-no effort to port between hardware systems. To validate the performance claim of this thesis design, the 'DSRP' was written as a complete instantiate of the 3R module and run in simulation. Results confirm it is both an efficient and scalable algorithm.

The DSRP is near completion, it further requires a helper-module class. Nevertheless and in contrast to all SRPs reviewed; the DSRP is the first decentralized, non-metamodule planner of arbitrary goal shapes to operate in real-time that can additionally direct the parallel motion of modules. It is also the first general purpose reconfiguration planner for the Superbot module.

6.2 Implications of Thesis

Groundwork laid by the DSRP is its high-level general methodology of constructing a state space of modules which can be searched to solve reconfigurations. This methodology is completely hardware-independent and can thus be applied to modules of any type. Additionally all principals of implementation that are hardware-dependent, such as state representation, state isomorphisms and the transition model can be applied to all modules discussed in chapter 2 and those yet to be built. This contrasts all SRPs reviewed in background chapter 3 with the possible exception of Graph Signature. This progresses current literature of SRPs to an almost-general solution. An approach to full generalization is discussed next section.

When the completed DSRP is coupled with static stability results from other team member projects we will have the world's first general-purpose 3R planner that can be directly implemented in hardware. This will be a full solution to the problem of autonomous self reconfigurations of a 3R-based SRR. At this point the team will additionally be able to address the topic of SRR applications by direct experimentation. By combining UWB radar attachments (another team member project) the robot will additionally be able to sense its surroundings opening up many new and exciting experimentation opportunities of autonomous operations.

The presentation of this paper including all concepts, descriptions and algorithms has been presented with the intension that the reader could emulate the DSRP if desired. It is the intention of this author that this approach can serve as a platform in which the design of a fully general reconfiguration planner can be extended from. Such a general planner holds large potential for the field of SRRs, able to solve reconfigurations of any lattice or hybrid based robots efficiently.

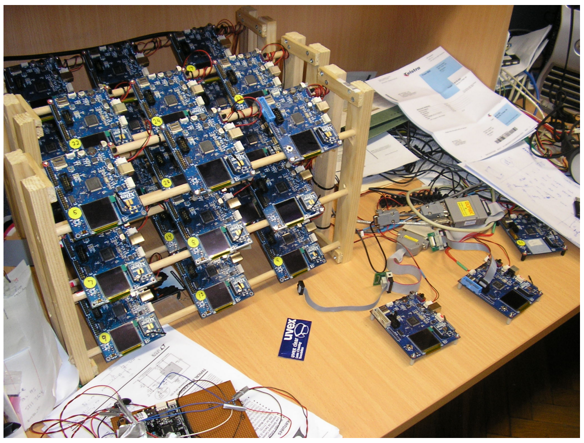


Figure 6.1: Hardware in the Loop Simulator

6.3 Future Work

6.3.1 Hardware Experiments

As part of the development of this thesis the author helped build the 'hardware in the loop' simulator seen in Fig. 6.1, a fully distributed system that forms the next stage of testing for the DSRP. The hardware simulator is a collection of STM32F microcontrollers from STMicroelectronics, each microcontroller representing one SRR module. For testing purposes it is essentially one step away from the real thing. It provides a realistic yet controlled computing architecture to validate all computation and memory assumptions made in computer simulation. The communication system is based on wireless ZigBee setup developed in house [15], to replicate packeted module-module message passing. To test the DSRP it first needs to be translated from Java in C++ to compile on the boards.

6.3.2 Roombots

After successful testing on the ‘hardware in the loop’ simulator, the next stage it to install the DSRP onto Roombot (sec. 2.3.6) modules manufactured by the ACFR’s collaborators at EPFL, Switzerland. Similar to the Superbot modules, Roombots offer a higher amount of isomorphisms and their uniform connector placements simplify planning significantly, and are thus an ideal module type to begin proper testing with.

6.3.3 Full Generalization

Generalization of a planner is the capability to solve reconfigurations for any arbitrary module design, not just for a particular one. Neither the CSRP or the DSRP have been fully generalized yet, both compute reconfigurations for the 3R module type. To generalize these planners, an automated means of developing the hard code in Transition Model (sec. 4.5.6 - 4.5.8) for any module design is required, as these are the only hardware dependent pieces of code not written dynamically. Both the CSRP’s and DSRP’s Transition Model code are exactly the same. So far the Collision Detection (sec. 4.5.7 routine is almost hardware independent though does assume the modules conform to cubic lattice architectures, however *Transition* (sec. 4.5.6) and *Translation* (sec. 4.5.8) are mostly hard-coded. This means the planner will operate for Superbot types modules, and also MTRAN type modules if all action options that actuate the central axle are denied (an MTRAN is just like Superbot except it does not have a central axle).

The challenge of generalization is complex and no planner has yet achieved it. To fully generalized the DSRP a template class is proposed which module specification could be entered into in such a way that would fully and uniquely describe a module’s functionality (geometry and kinematics). From here, all instances of hard code within Transition Model need to be re-written as dynamic code that use only the information presented on the in-putted template. With routines Collision Detection, Transition and Translation updated the DSRP will be fully generalized, able to operate on any module hardware.

Appendix A

Index to Multimedia Extensions

Table A.1: Index to Multimedia Extensions

Ext.	Type	Filename	Description
1	Video	SingleLocomotion.wmv	mobile module moves without helper
2	Video	CoordinatedLocomotion.wmv	helper aids mobile module's locomotion
3	Video	LineToRing.wmv	CSRP reconfiguration
4	Video	LineToBox.wmv	CSRP reconfiguration
5	Video	LineToSuperbotman.wmv	CSRP reconfiguration
6	Video	LineToSidestack.wmv	CSRP reconfiguration
7	Video	RingToLine.wmv	CSRP failed attempt
8	Video	BoxToLine.wmv	CSRP reconfiguration
9	Video	SuperbotmanToLine.wmv	CSRP reconfiguration
10	Video	SidestackToLine.wmv	CSRP reconfiguration
11	Video	DSRPExperimentD7.wmv	DSRP reconfiguration

Appendix B

Extra Routines

Algorithm B.1 Encode State

```
Input (State) state
Initialize $code \leftarrow 0$
$code = 108\times$((moduleDirectionSign+1)/2) // [-1,1]
$+36\times$moduleDirection // [0,1,2]
$+18\times$encodeAxisDirection(moduleDirection,connPartAxisDirn) // [0,1]
$+9\times$encodeAxisDirection(moduleDirection,unconnPartAxisDirn) // [0,1]
$+3\times$connectedPartRotationIndex // [0,1,2]
$+1\times$unconnectedPartRotationIndex // [0,1,2]
**return** (byte) code
```

Algorithm B.2 Decode State

```
1: Input (byte) code
2: Initialize (State) state
3: state.moduleDirectionSign = $2 \times (code/108) - 1$
4: $code = remainder(code, 108)$
5: state.moduleDirection = $code/36$
6: $code = remainder(code, 36)$
7: state.connPartAxisDirn = decodeAxisDirn(moduleDirection, ($code/18$))
8: $code = remainder(code, 18)$
9: state.unconnPartAxisDirn = decodeAxisDirn(moduleDirection, ($code/9$))
10: $code = remainder(code, 9)$
11: state.connectedPartPossibleVectorsIndex = $code/3$
12: $code = remainder(code, 3)$
13: state.unconnectedPartPossibleVectorsIndex = $code$
14: **return** (State) state
```

Glossary

Bipartite A module consisting of two parts, usually linked together by an axle

Chain (Module) A module architecture type that allows module configuration values (position, direction, axle rotation etc.) to be continuous (not discrete)

Complete (Algorithm) An algorithmic property; if a solution exists, then the algorithm always finds it

Complexity Cost analysis of a program's execution. Time complexity refers to relationship between execution time and the number of modules in a system, and space complexity refers to the relationship between memory required and the number of modules in a system. *Complexity*, when used as an isolated term is this report, refers to *both* time and space complexity

Concavity A concavity is a lattice-cell that is adjacent to two or more non-module modules

Configuration (Module) The description of a module in 3D space including its: i) position, ii) orientation and iii) rotation about its degrees of freedom

Configuration (Modular Cluster/Robot) A particular arrangement of connectivity between independent modules

Configuration Space (Modular Cluster/Robot) The set of unique configurations a modular cluster can assume

Connectivity Connectivity is the physical linkage of modules. Global connectivity refers to a modular robot that is structurally a single piece by virtue of the various individual connections linking composing modules. Global connectivity is a binary statement; a modular robot is either globally connected (one piece), or it is not (meaning it is in multiple separated clusters)

Converter Module see *Helper Module*

Correct (Algorithm) An algorithmic property; if a algorithm finds a solution, then that solution is always a correct solution

Cousin State When a state links to another state located within a different module's LSN (Local State Network), both states are referred to as *cousin* states

Degree of Freedom Means in which a module can alter its shape, usually an actuated axles or hinges that rotate one part of the module relative to another part

Global connectivity see *Connectivity*

Grid see *Lattice (Grid)*

Hard Code A fixed coded object, such as a data or routine, stored in the program's source code. This object is final, and cannot be dynamically updated or automatically generated by the program. Hard code often has limited use and is usually not desired over programs that can automatically update the object to suit changing circumstances

Helper Module A module that will reconfigure itself to aid a surface-moving modules move by picking it up at one location, and placing it down in another, without detaching itself from the module cluster. All helper modules are *structural* modules

Hybrid (Module) A module design which can act as either a chain or lattice architecture module

Lattice (Grid) A hypothetical framework in 3D space specifying regular polyhedral cell locations packed together (e.g. honeycomb lattice of hexagon cells, or graph paper of square cells)

Lattice (Module) A module architecture type that limits module configuration values (position, direction, axle rotation etc.) to be discrete

Mobile Module see *Surface Moving Module*

Mobilized The event of a *structural* module changing its class into a *mobile* module

Module A robotic 'building block' of modular robots. Usually has at least 2 connecting faces to be able to connect with other modules. May contain a power source(s), microchip(s), degree of freedom(s), sensor(s), etc. See sec.2.2 for details.

Module Cluster Multiple connected modules forming a single modular structure.

Monopartite A module consisting of one part

Navigation Function A function that assigns numeric values of desirability to different states or connecting surfaces. Mobile modules can consider these values when greedily deciding between which states to progress to as part of their informed locomotion between two locations. Usually a higher value represents higher desirability, a goal location or state is given a desirability of zero, and all surrounding states/connecting faces will have negative values, the magnitude of which represents their action-distance to the goal.

Native Kinematics The kinematic abilities of a module specific to its design

Neighbor State (CSRP) If a module in a certain state was to take any action it will result in another state. Both these states are referred to as *neighbor* states

Neighbor State (DSRP) When a state within a module's LSN (Local State Network) links to another state also within the same module's LSN, both states are referred to as *neighbor* states

Reconfiguration A transition between two configurations by a series of atomic movements

Reconfiguration Planner Optimal algorithm that minimizes a measure of interest such as the number of steps,time or power required to reach a final configuration

Searchable Space see *State Space (Modular Cluster/Robot)*

Space Complexity see *Complexity*

State (Module) A description of the functional state of a module. Often multiple module-configurations can be in the same *state* being functionally (geometrically and kinematically) identical.

State (Modular Cluster/Robot) A set of configurations that are functionally (geometrically and kinematically) identical

State Network A network of unique state-nodes. All states that exist in a module state space exist in its state network as nodes which are interlinked by actions the module can take. Valid actions a module can take result in a change of state of the module, thus such an action provides a direct link between two state-nodes in a modules state network. A state network can exist for just one mobile module to plan its movements via a navigation function. It can also be used in the context of a cluster of modules and has the same meaning, where each state represents a unique states of the modular cluster.

State Space (Modular Cluster/Robot) The set of unique states a modular cluster can assume

Static Module A module that does not move at all (unless mobilized). Thus it is not a mobile module nor a helper module, but part of the robot's static structure of modules remain in their positions. All static modules are *structural* modules

Strictly Decentralized A type of planning algorithm that restricts module-module communication to only a few module lengths. This is distinct to non-strictly decentralized planners (or just '*decentralized planners*') which still execute in parallel as decentralized algorithms do however also have the option of broadcasting information to any other module in the modular cluster no matter how far away

Structural Module Either a *helper* module or a *static* module in a robot's cluster. If a module it not a *mobile* module then it must be a *structural* module.

Substate The elements of a *state* that concern just one of a *bipartite* module's parts. Used to represent the 'state' of one module part

Superstate A duel set of *states* that describe both a mobile-module and a helper-module

Surface Moving Module A mobilized module traveling over the surface of static modules which form the bulk of a modular cluster

Thread A portion of a program that can run independently of and concurrently with other portions of the program - [www.answers.com]

Time Complexity see *Complexity*

Bibliography

[1] M. Asadpour, A. Sproewitz, A. Billard, P. Dillenbourg, and A. Ijspeert, *Graph signature for self-reconfiguration planning*, IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS-2008), September 2008, pp. 863–869.

[2] M. Ashley-Rollman, M. De Rosa, S. Srinivasa, P. Pillai, S. Goldstein, and J. Campbell, *Declarative programming for modular robots*, Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS '07, October 2007.

[3] SD Card Association, *Sd technology overview*, `http://www.sdcard.org/developers/tech/`, 2008, last visited 1 Nov. 2009.

[4] A. Barto and S. Mahadevan, *Recent advances in hierarchical reinforcement learning*, Discrete Event Dynamic Systems **13** (2003), no. 1-2, 41–77.

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

[6] CMU Biorobotics, *Modular snake robots*, `www.modsnake.com`, July 2009, last visited 1 Nov. 2009.

[7] R. Braeunig, *Planetary spacecraft*, `http://www.braeunig.us/space/planet.htm`, 2008, last visited 1 Nov. 2009.

[8] A. Castano, A. Behar, and P. Will, *The conro modules for reconfigurable robots*, IEEE/ASME Transactions on Mechatronics, December 2002, Issue 4, pp. 403–409.

[9] G. Chirikjian, *Kinematics of a metamorphic robotic system*, IEEE Intl. Conf. on Robotics and Automation, 1994, pp. 449–455.

[10] D. Christian, *Maps of time: An introduction to big history*, Univeristy of California Press, Ltd, 2004.

[11] CMU, *Claytronics group publications listed by year*, `http://www.cs.cmu.edu/~claytronics/publications/index.html`, 2008, last accessed 25 Sept 2009.

[12] T. Dietterich, *Hierarchical reinforcement learning with the maxq value function decomposition*, Journal of Artificial Intelligence Research **13** (1999), 227–303.

[13] S. Finch, *Mathematical constants*, vol. 94, Cambtidge University Press, 2003.

[14] R. Fitch and Z. Butler, *Million module march: Scalable locomotion for large self-reconfiguring robots*, The International Journal of Robotics Research **27** (2008), no. 3-4, 331–343.

[15] R. Fitch and R. Lal, *Experiments with a zigbee wireless communication system for self-reconfiguring modular robots*, IEEE Int. Conf. on Robotics and Automation, 2009. ICRA '09 (Kobe International Conference Center, Kobe, Japan), May 2009, pp. 1947–1952.

[16] T. Fukuda and S. Nakagawa, *Approach to the dynamically reconfigurable robotic system*, Journal of Intelligent and Robotic Systems **1** (1987), 55–72.

[17] ______, *Dynamically reconfigurable robotic system*, Robotics and Automation; 1988 IEEE International Conference (Philadelphia, PA, USA), vol. 3, April 1988, pp. 1581–1586.

[18] T. Fukuda, S. Nakagawa, Y. Kawauchi, and M. Buss, *Self organizing robots based on cell structures - ckbot*, Intelligent Robots; IEEE International Workshop, November 1988, pp. 145–150.

[19] S. Goldstein, J. Campbell, and T. Mowry, *Programmable matter*, IEEE Computer **38** (2005), no. 6, 99–101.

[20] AIST Intelligent Systems Institute Distributed System Design Group, *Modular robot takes its first step: Modular transformer (m-tran)*, `http://staff.aist.go.jp/e.yoshida/test/index-e.htm`, April 2003, last visited 1 Nov. 2009.

[21] N. Inou, K. Minami, and M. Koseki, *Group robots forming a mechanical structure-development of slide motion mechanism and estimation of energy consumption of the structural formation*, IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA) (Kobe, Japan), July 2003, pp. 874–879.

[22] USC Information Sciences Institute, *1 km rolling track*, `http://www.isi.edu/robots/media-superbot.html`, June 2007, last visited 1 Nov. 2009.

[23] A. Kamimura, S. Murata, E. Yoshida, H. Kurokawa, K. Tomita, and S. Kokaji, *Self-reconfigurable modular robot - experiments on reconfiguration and locomotion*, IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS-2001), vol. 1, October-November 2001, pp. 606–612.

[24] K. Kotay, D. Rus, M. Vona, and C. McGray, *The self-reconfiguring robotic molecule: Design and control algorithms*, IEEE Int. Conf. on Robotics & Automation (ICRA), May 1998, pp. 424–431.

[25] H. Kurokawa, K. Tomita, E. Yoshida, S. Murata, and S. Kokaji, *Motion simulation of a modular robotic system*, IEEE Int. Conf. Industrial Electronics, Control and Instrumentation (IECON-2000), Mechanical Engineering Laboratory, AIST, MITI, October 2000, pp. 2473–2478.

[26] S. Murata, H. Kurokawa, and S. Kokaji, *Self-assembling machine*, IEEE Int. Conf. on Robotics & Automation (ICRA94) (San Diego, CA, USA), vol. 1, May 1994, pp. 441–448.

[27] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji, *A 3-d self-reconfigurable structure*, IEEE Int. Conf. on Robotics & Automation (ICRA), vol. 1, May 1998, pp. 432–439.

[28] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji, *M-tran: self-reconfigurable modular robotic system*, IEEE/ASME Transactions on Mechatronics (Yokohama, Japan), vol. 7, Tokyo Inst. of Technol., December 2002, issue 4, pp. 431–441.

[29] S. Murata, E. Yoshida, K. Tomita, H. Kurokawa, A. Kamimura, , and S. Kokaji, *Hardware design of modular robotic system*, IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS00) (Takamatsu, Japan), 2000, pp. 2210–2217.

[30] NASA, *Mars rovers braving severe dust storms*, `http://www.jpl.nasa.gov/news/news.cfm?release=2007-080`, July 2007, last visited 1 Nov. 2009.

[31] ______, *Dust storm cuts energy supply of nasa mars rover spirit*, `http://marsrovers.nasa.gov/newsroom/pressreleases/20081110a.html`, November 2008, last visited 1 Nov. 2009.

[32] ______, *Space shuttle and international space station*, `http://www.nasa.gov/centers/kennedy/about/information/shuttle_faq.html#10`, February 2008, last visited 1 Nov. 2009.

[33] E. Østergaard, K. Kassow, R. Beck, and H. Lund, *Design of the atron lattice-based self-reconfigurable robot*, Autonomous Robots **21** (2006), no. 2, 165–183.

[34] K. Prevas, C. Unsal, M. Efe, and P. Khosla, *A hierarchical motion planning strategy for a uniform self-reconfigurable*, IEEE International Conference on Robotics and Automation (ICRA), vol. 1, May 2002, pp. 787–792.

[35] T. Regge and R. Zecchina, *Combinatorial and topological approach to the 3d ising model*, Journal of Physics A **33** (2000), 741–761.

[36] M. De Rosa, S. Goldstein, P. Lee, J. Campbell, and P. Pillai, *Scalable shape sculpting via hole motion: Motion planning in lattice-constrained module robots*, Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA '06), May 2006.

[37] ———, *Programming modular robots with locally distributed predicates*, Proceedings of the IEEE International Conference on Robotics and Automation ICRA '08, 2008.

[38] D. Rus and M. Vona, *Crystalline robots: Self-reconfiguration with compressible unit modules*, Autonomous Robots **10** (2001), no. 1, 107–124.

[39] LLC Practical Robotic Services, *Unimate robots*, `http://www.prsrobots.com/unimate.html`, last visited 1 Nov. 2009.

[40] W. Shen, M Krivokon, H. Chiu, J. Everist, M. Rubenstein, and J. Venkatesh, *Multimode locomotion via superbot reconfigurable robots*, Autonomous Robots **20** (2006), no. 2, 165–177.

[41] A. Sproewitz, A. Billard, P. Dillenbourg, and A. Ijspeert, *Roombots–mechanical design of self-reconfiguring modular robots for adaptive furniture*, Proc. 2009 IEEE ICRA, to appear, May 2009.

[42] A. Sproewitz, R. Moeckel, J. Maye, and A. Ijspeert, *Learning to move in modular robots using central pattern generators and online optimization*, Int. J. Rob. Res. **27** (2008), no. 3-4, 423–443.

[43] E. Stofan, C. Elachi, J. Lunine, R. Lorenz, B. Stiles, K. Mitchell, S. Ostro, L. Soderblom, C. Wood, H. Zebker, S. Wall, M. Janssen, R. Kirk, R. Lopes, F. Paganelli, J. Radebaugh, L. Wye, Y. Anderson, M. Allison, R. Boehmer, P. Callahan, P. Encrenaz, E. Flamini, G. Francescetti, Y. Gim, G. Hamilton, S. Hensley, W. Johnson, K. Kelleher, D. Muhleman, P. Paillou, G. Picardi, F. Posa, L. Roth, R. Seu, S. Shaffer, S. Vetrella, and R. West, *The lakes of titan*, Nature **445** (2007), 61–64.

[44] R. Sutton and A. Barto, *Reinforcement learning: An introduction*, MIT Press, 1998.

[45] Duke University, *Self-reconfigurable robots: Papers*, `www.cs.duke.edu/~sgs/robots/papers.php`, December 2008, last visited 1 Nov. 2009.

[46] C. Ünsal and P. Khosla, *Mechatronic design of a modular self-reconfiguring robotic system*, IEEE International Conference on Robotics & Automation (ICRA) (San Francisco, CA, USA), vol. 2, April 2000, pp. 1742–1747.

[47] P. Vanhavskaya, L. Kaelbling, and D. Rus, *Learning distributed control for modular robots*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), vol. 3, 2004, pp. 2648–2653.

[48] www.oloscience.com, *Snake robot in the water*, `http://www.youtube.com/watch?v=vn2Pb_Kh8Pk`, June 2008, last visited 1 Nov. 2009.

[49] M. Yim, K. Roufas, D. Duff, Y. Zhang, C. Eldershaw, and S. Homans, *Modular reconfigurable robots in space applications*, Auton. Robots **14** (2003), no. 2-3, 225–237.

[50] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. Chirikjian, *Modular self-reconfigurable robot systems: Challenges and opportunities for the future*, IEEE Robotics and Automation Mazazine (2007), 43–52.

[51] M. Yim, Y. Zhang, and D. Duff, *Modular robots*, IEEE Spectrum (2002), 30–34.

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

[53] E. Yoshida, S. Murata, A. Kamimura, K. Tomita, H. Kurokawa, and S. Kokaji, *A self-reconfigurable modular robot: Reconfiguration planning and experiments*, The International Journal of Robotics Research **21** (2002), no. 10, 903–916.

[54] E. Yoshida, S. Murata, S. Kokaji, K. Tomita, and H. Kurokawa, *Micro self-reconfigurable robotic system using shape memory alloy*, Distributed Autonomous Robotic Systems **4** (2000), 145–154.

[55] V. Zykov, E. Mytilinaios, M. Desnoyer, and H. Lipson, *Evolved and designed modular robotics systems capable of self-reproduction*, IEEE Trans. Robotics (Mech. & Aerosp. Eng., Cornell Univ., Ithaca, NY), vol. 23, April 2007, Issue 2, pp. 308–319.