HIGH-LEVEL CONTROL OF MODULAR ROBOTS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Sebastian A. Castro

January 2012

ABSTRACT

Reconfigurable modular robots can exhibit different specializations by rearranging the same set of parts comprising them. Actuating modular robots can be complicated because of the many degrees of freedom that scale exponentially with the size of the robot. Effectively controlling these robots directly relates to how well they can be used to complete meaningful tasks.

This paper discusses an approach for creating provably correct controllers for modular robots from high-level tasks defined with structured English sentences. While this has been demonstrated with simple mobile robots, the problem was enriched by considering the uniqueness of reconfigurable modular robots. These requirements are expressed through *traits* in the high-level task specification that store information about the geometry and motion types of a robot.

Given a high-level problem definition for a modular robot, the approach in this paper deals with generating all lower levels of control needed to solve it. Information about different robot characteristics is stored in a library, and two tools for populating this library have been developed. The first approach is a physics-based simulator and gait creator for manual generation of motion gaits. The second is a genetic algorithm framework that uses *traits* to evaluate performance under various metrics. Demonstration is done through simulation and with the CKBot hardware platform.

BIOGRAPHICAL SKETCH

Sebastian A. Castro was born November 3, 1988 in Santiago, Chile. In 1999 he moved to Kingston, Jamaica where he attended the Campion College secondary school and graduated in 2006 with a Communication Studies award and an Academic Excellence award. He then attended Cornell University for his undergraduate studies, where he graduated Magna Cum Laude with a Bachelor of Science in Mechanical and Aerospace engineering in 2010. During his undergraduate career, Sebastian worked on Heating, Ventilation and Air Conditioning controls with the 2009 Solar Decathlon project team and on an autonomous wheeled vehicle controls team under Dr. Ephrahim Garcia. Since June 2010, Sebastian has served as a research assistant at the Autonomous Systems Laboratory at Cornell University, advised by Dr. Hadas Kress-Gazit.

ACKNOWLEDGEMENTS

I would like to firstly thank Dr. Hadas Kress-Gazit and Dr. Mark Campbell for granting me the opportunity and guidance to conduct my research towards a Master of Science degree. Dr. Kress-Gazit and Dr. Campbell are not only co-directors of the Autonomous Systems Laboratory and the Chair and Co-chair (respectively) of my defense committee, but they have also taught me much of what I know about the disciplines of Robotics and Control Systems. These areas just so happen to be my favorite areas in Mechanical Engineering.

My colleagues at the Autonomous Systems Laboratory are very important to me. I would like to extend my special gratitude to Sarah Koehler, my awesome research partner for two whole semesters of school. Cameron Finucane, Jim Jing, Eric Sample and Luis Zamora were also invaluable to my progress, always there to save my skin with any bottlenecks that I encountered.

Dr. Mark Yim and Jimmy Sastra from the University of Pennsylvania also have my sincerest thanks. They provided me with all the support I needed regarding the CKBot modular robot platform, and even welcomed me to Philadelphia for a few days to teach me how to work with their robots so I could produce an example for submission to the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems.

Finally, I would like to thank my wonderful friends and family for the sole fact that they are my friends and family; the least they deserve is to be mentioned here for collectively (through whichever of many microscopic ways) steering the world into the state that it's currently in right now. Without any of you, who knows if I would even be writing these acknowledgments right now?

TABLE OF CONTENTS

1 Introduction **1**

2 Paper I: High-Level Control of Modular Robots **2**
2.1 Introduction 2
2.2 Related Work 4
2.3 Problem Formulation 6
2.4 Background 9
2.4.1 Modular Robots - CKBot 9
2.4.2 Modular Robots - Configurations and Gaits 10
2.4.3 High-Level Control - LTLMoP 12
2.5 Approach 14
2.5.1 Traits 14
2.5.2 Guaranteeing Correct Control 16
2.6 Populating the Library 18
2.6.1 Manual Population: Gait Creator 19
2.6.2 Automatic Population: Genetic Algorithms 20
2.7 Results 25
2.7.1 Gait Creator 25
2.7.2 Genetic Algorithms 26
2.8 Example: Rescue Robot 28
2.8.1 Task Specification 28
2.8.2 Execution of Task Specification 30
2.9 Conclusions and Future Work 30
2.10 Acknowledgments 32

Bibliography **33**

LIST OF TABLES

2.1 Sample List of Traits and Configuration-Gait Pairs 15

LIST OF FIGURES

2.1 Example Task Specification in structured English. 9
2.2 An Ethernet-powered "Tee" configuration of CKBot modules with Vicon markers (left) and simulated 25-module "Hexapod" configuration (Right). 10
2.3 A simulated CKBot configuration with port-adjacency matrix. T_W^0 is the **base transformation matrix** dictating what translation and rotation the robot is spawned with, and $\vec{v_f}$ is the **forward vector** used to keep track of the orientation of the robot on the 2D ground plane. 11
2.4 Automaton schematic for Example 1. 18
2.5 Tripod Configuration in the Gait Creator. The module being controlled is highlighted in red. 19
2.6 Manually generated Slinky gait. 26
2.7 Best Orientation time history for Cross configuration under trait "TurnInPlaceLeft" (Top). Screenshots of configuration running the resulting gait (Lower). 27
2.8 Best base module height time history for Snake configuration under the traits "Tall" and "TurnRight"(Top). Screenshots of configuration running the resulting gait (Lower). 27
2.9 Task Specification in structured English. 28
2.10 Workspace for the task specification. 29
2.11 Simulation screen shots. The images, from top to bottom, left to right, show: a) The default Snake.crawl configuration-gait pair moving in the environment, b) The Loop.roll configuration-gait pair in the Trench, c) The Tripod.crawl configuration-gait pair when air_raid is active and d) Snake.crawl returning to the Safehouse while carrying_person is true. . 31
2.12 Hardware screen shots showing CKBot powered via Ethernet, with Vicon markers for pose information. The Tripod.crawl configuration-gait pair moving in the Trail region as air_raid is being sensed (left) and the default Snake.crawl configuration-gait pair (right). 31

Chapter 1

Introduction

This thesis is comprised one journal-style paper which summarizes the approach for high-level control of modular robots. This paper, presented in Chapter 2, outlines a high-level control approach for reconfigurable modular robots as well as a modular robot simulator and two methods to create motion for different shapes of modular robots. As it will be discussed, this paper builds on the author's submission to the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems [8].

The content in Chapter 2 is, for the most part, identical to the September 20th, 2011 draft of the final journal paper. The only exception is that the journal submission contains an accompanying video extension to further show experimental results. All references to this video have been omitted in the version below. The abstract for this journal paper is identical to the abstract for this thesis.

Chapter 2

Paper I: High-Level Control of Modular Robots

2.1 Introduction

Instructing robots to safely and correctly perform complicated tasks is a prevalent challenge in robotics. It is even more challenging to achieve this if with the intent to command robots in a way that is easy and expressive for humans. Establishing methods for humans to more naturally communicate with robots is useful since it removes the need for operators to be robot specialists. This research area of high-level robot control carries the ultimate goal of having humans convey tasks to robots as they would to another human, with the purpose of solving meaningful and difficult problems.

Modular robots are robots composed from several identical, or similar, building blocks (or *modules*) that can be rearranged into different geometries. In this approach modular robots are categorized for problem solving with qualitative labels known as *traits*. For example, a high-level task specification can require a robot to have a "low" motion profile when navigating in a region with a low clearance or move "fast" when in a dangerous zone that requires quick traversal. The words "low" and "fast" are traits that describe the state of a modular robot. In a task specification, traits can be activated and deactivated based on the location of the robot or as a reaction to the environment. Depending on which traits are active, a modular robot may automatically reconfigure and/or change its motion qualities.

Existing high-level control techniques for non-reconfigurable robots were extended for use with modular robots. A high-level control problem may now contain specifi-

cations on the structure and motion profile of a robot by using qualifying words rather than through explicit commands or rules. Two approaches to qualitatively populate a database of modular robots were developed; the population of this database is important as it strengthens the search space for automatic reconfiguration of modular robots. Both of these tools were designed to create motion profiles for predefined geometries of modular robots, which overcomes the difficulty of analyzing the dynamics of modular robots. The first approach is a manual interface which requires human input, and the second is an automatic process which applies Genetic Algorithms [12] that are scored using these same qualifying traits as user input.

This approach is novel because most prior work with modular robots involves relatively specific goals. Refer to Section 2.2 for details on pertinent modular robotics work of this nature. There are multiple layers of control for reconfigurable modular robots, from high-level structured English instructions to automatic configuration and motion selection to frameworks for generation of low-level control for several robot shapes. In other words, this paper captures a holistic solution for using modular robots to solve problems that humans can deem useful and significant.

This paper builds directly on [8] with more theoretical detail and examples. In addition, a Genetic Algorithm framework (mentioned in Section 2.6.2) was developed to automate motion generation for modular robots. The paper is structured as follows: Section 2.2 outlines some of the relevant previous work. Section 2.3 describes the problem that is being solved. Section 2.4 provides background regarding the general approach to generating correct control from high-level specifications and the hardware platform used in the experiments. Section 2.5 discusses the approach of incorporating traits into the task description and sections 2.6 and 2.7 demonstrate gait generation techniques. Section 2.8 outlines the approach in simulation and with a hardware experiment through a

sample task specification scenario. The paper concludes with future research directions in Section 2.9.

2.2 Related Work

There has recently been much research with modular robots due to their versatility and adaptability in new scenarios and the low cost of replacing individual modules [35]. The focus of this paper is on *chain type* modular robots [36]. This type of modular robot does not reconfigure by dividing into several substructures; that is, all modules remain attached to one another such that there is always one robot [37]. Other types of modular robots include *lattice type*, where modules are arranged in regular structures and can move relative to their neighbors and *mobile type*, which use the environment to assist their motion.

Reconfiguration planning with modular robots has been a common topic of investigation. The work in [17] presents the design of a modular robot and manual configuration and motion interface with a simulator; it deals mostly with self-reconfiguration experiments and motion planning rather than a high-level approach involving transformation of robots. Other work involving simulation and reconfiguration of modular robots includes [32], where a gripper made of modular robots is used to navigate an obstacle-filled environment and to pick up other modules. [26] and [9] present mathematical motion planning approaches for optimal reconfiguration of modular robots by using cost metrics. Reconfiguration planning for lattice-type modular robots has been investigated as in [38]. The approach in this paper treats reconfiguration at a higher level; rather than planning the actual process of reconfiguring a robot, an algorithm searches for configurations that meet qualitative user specifications.

Genetic Algorithms [12] are employed in robotics for motion planning (see, for ex-

ample, [6]). The approach in this paper is related to [25, 24, 16] where Genetic Algorithms are used to optimize modular robot motion using biologically-inspired Central Pattern Generators (CPG) techniques [13]. The fitness functions in [25, 24] are only dependent on the total distance traveled by the robot, and those in [16] additionally introduce penalties on deviating from straight lines and using too much motor energy. On the other hand, this paper describes a way to utilize traits to design fitness functions that evaluate the motion of a modular robot at a high level.

Recent work in high-level control of mobile robots has sought to automatically generate controllers from instructions that are more intuitive for humans. Temporal logic (in work such as [21, 19, 7, 18, 33]) or structured language (as in [20]) are used to define high-level task specifications. These approaches, in brief, convert a real-life robotics problem into a hybrid system consisting of multiple layers of abstraction. The upper layer is the problem consisting of logical propositions, and on the lower layer there is continuous execution of the robot behavior. Occasionally, this continuous execution results in instantaneous changes in the binary propositions of the upper layer. Some of these approaches (for example, [21, 33]) also involve the use of robot sensors to react to the environment. That is, information gathered from sensors is converted into binary propositions present in the task specification. Other research explores the probabilistic analysis of imperfect actions [23] or imperfect information about the environment [15]; however, the work in this paper currently assumes perfect sensing and actions in the discrete abstraction.

This approach (along with [8]) builds on the work in [21, 20], using the high-level control framework in [11]. Here a task specification is parsed into structured English constructed around a subset of Linear Temporal Logic (LTL) ([10]). This grammar allows for conditional statements (e.g. “If you are sensing *lowFuel* then visit *gasStation*”),

safety requirements (e.g. "Always not *dangerRegion*) and non-projective locative prepositions such as "between", "within *distance* of", and more. The details of this grammar are discussed in [20, 22].

2.3 Problem Formulation

A few definitions which describe modular robots at a high level are required. A modular robot is composed of a specific arrangement of modules known as a **configuration**. A modular robot configuration moves through **gaits**, which are repeatable motions of each module in the configuration. An individual configuration can have many gaits; for example, a straight line of modules can move by *crawling* like a worm or by *folding over* in a slinky-like fashion. A **configuration-gait pair** $g(t) \in G$ of a robot is defined as the configuration and type of gaits (crawling, rolling, etc.) it adopts at time t. G is the set of all configuration-gait pairs available for a specific problem. The configuration-gait pair "Snake.crawl", for instance, denotes a *Snake* configuration using a set of *crawling* motion gaits. Note that a configuration-gait pair involves a single robot configuration, but can have multiple gaits. "Snake.crawl" contains 3 crawling gaits for forward locomotion, clockwise turning and counterclockwise turning.

In order to describe the properties of different configuration-gait pairs, a set T of **traits** is defined. Each trait $T_i \in T$ is an English word (or phrase) which describes the geometry or motion profile of a modular robot. For example, a modular robot that is narrow in shape and is only able to move forward or backward in a straight line may be assigned the traits "narrow" and "1D_motion" to describe it. Traits are used as logical propositions in a task specification; the configuration-gait pair $g(t)$ of a robot depends on which traits are active at any given time t.

A modular robot moves in an environment P such that its trajectory in continuous

time is $p(t) \in P, \forall t > 0$. The robot may be equipped with sensors (for example, cameras or sonar) and actuators (for example, speakers or lights). Specifications involving location and sensing of the environment, actuators and traits are then synthesized into provably correct control of modular robots. The robot and its environment are described using binary prepositions in the structured English grammar described in [20, 22]. The grammar consists of the following items:

- Set of **sensors** X corresponding to information the robot can obtain about the environment through sensors.
- Set of **actions** A that the robot can perform. The set of active actions at a given time t is defined as $a(t) \in 2^A$ where 2^A denotes the power set of A.
- Set of **regions** R corresponding to regions of interest in the specified environment.
- Set of **traits** T that describe properties of a modular robot. Traits are distinguished by adding a prefix "T_" to each trait. As discussed in Section 2.5, a mapping $\Gamma : T \to 2^G$ is defined such that $\Gamma(T_i)$ is the set of all possible configuration-gait pairs that satisfies a trait $T_i \in T$.

The problem addressed in this paper is the following:

Problem 1 [High-level Control of Modular Robots] Given a modular robot operating in a known workspace P and a high-level task specification S expressed in structured English using the sets X, A, R, T, construct (if possible) a controller so that the robot's resulting trajectories $p(t)$, actions $a(t)$ and configuration-gait pairs $g(t)$, satisfy the system specification S in any admissible[1] environment, from any allowable initial state.

Example 1. The following example shows how a simple problem can be formulated using this construction. Refer to the line numbers in the task specification S shown in Fig. 2.1. Consider a simple environment consisting of two regions *Indoors* and *Outdoors*. The robot is additionally equipped with a *danger* sensor and a *Shrink* actuator. The traits used for this example are *legged* and *narrow*. The abstraction of the problem is shown below.

- $X = \{\text{danger}\}$
- $A = \{\text{Shrink}\}$
- $R = \{\text{Outdoors}, \text{Indoors}\}$
- $T = \{\text{T\_legged}, \text{T\_narrow}\}$

A modular robot begins *Outdoors* (line 1) with the assumption that every initial state of the environment is false(line 2). It will infinitely often cycle between visiting *Outdoors* and *Indoors* (lines 3-4), ensuring to be in a "legged" configuration-gait pair when *Indoors* (line 5). In addition, the robot is equipped with a sensor that can sense *danger* in the environment and will *Shrink* when this is sensed. When the robot is shrinking, it must be in a "narrow" configuration-gait pair (line 6).

[1] As discussed in [21], the specifications may include assumptions about the behavior of the environment, for example "*lowFuel* cannot happen in *dangerRegion*". An admissible environment is one that satisfied all the assumptions.

The trait mappings Γ and configuration-gait pair set G for this problem are:

- $\Gamma(\text{legged}) = \{\text{Tripod.crawl}, \text{Biped.splits}\}$
- $\Gamma(\text{narrow}) = \{\text{Loop.roll}, \text{Snake.crawl}\}$
- $G = \Gamma(\text{legged}) \cup \Gamma(\text{narrow})$

```
1: Robot starts in Outdoors
2: Env starts with false
3: Visit Outdoors
4: Visit Indoors
5: Do T_legged if and only if you are in Indoors
6: Do Shrink and T_narrow and only if you are
   sensing danger
```

Figure 2.1: Example Task Specification in structured English.

2.4 Background

2.4.1 Modular Robots - CKBot

The Connector Kinetic roBot (CKBot) developed by Yim et al. [27] is used to demonstrate this work. CKBot modules are single degree-of-freedom cubes actuated by rotational servo motors that each contain 7 infrared receiver-transmitter pairs on 4 of their 6 faces for connecting to other modules. There are 40 ways to connect any two CKBot modules, which provides extensive customizability for creating larger configurations. Additionally, a 3D physics-based CKBot simulator [8] is used in order to easily create configurations without being limited by the availability and functionality of hardware.

Figure 2.2 shows the hardware with two different power and communication attachments.

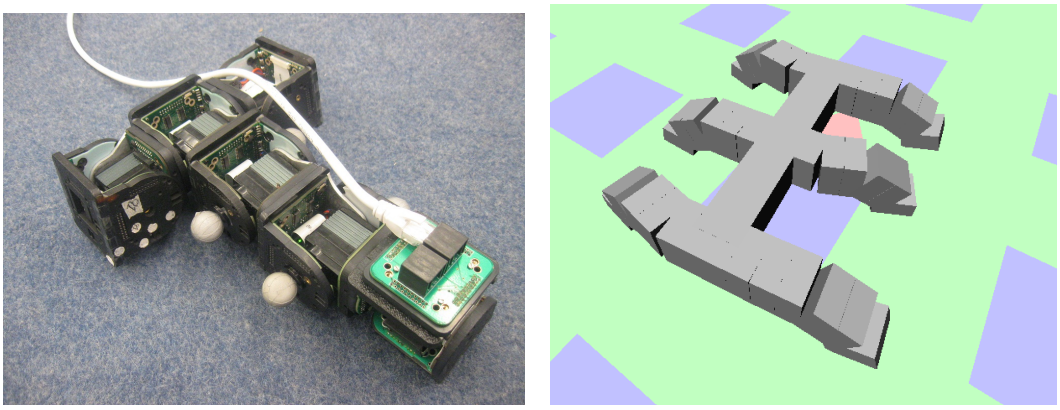

Figure 2.2: An Ethernet-powered "Tee" configuration of CKBot modules with Vicon markers (left) and simulated 25-module "Hexapod" configuration (Right).

Prior work with CKBot has mostly covered dynamics or kinematics of specific robot configurations. For example, CKBot has been demonstrated to execute dynamic rolling gaits of loops of modules [31] or legged motion aided with compliant legs [30]. Other research has involved CKBot modules connected to form high degree-of-freedom robot arms [5], with modified continuous-rotation servo modules with wheels for locomotion of robots. The focus on high-level control using CKBot is, to the best of the authors' knowledge, a novel approach.

2.4.2 Modular Robots - Configurations and Gaits

A **configuration** is the arrangement of modules of a modular robot. Configurations are usually represented as graphs of parent-child module connectivity. CKBot configurations are defined using a **port-adjacency matrix** [27] to depict connections between infrared receiver-transmitter pairs. For n modules, this matrix $\mathscr{M} \in \mathbb{R}^{n \times n}$ is a (usually

sparse) square matrix where each non-zero symmetric pair $\left(\mathscr{M}_{ij}, \mathscr{M}_{ji}\right), i \neq j$ denotes connectivity between port $\mathscr{M}_{ij}$ of module i and port $\mathscr{M}_{ji}$ of module j. Figure 2.3 shows how a modular robot is described using a port-adjacency matrix; for example, there is a connection between Port 3 of Module 0 and Port 5 of Module 1 as seen in the upper left symmetric pair of the matrix.

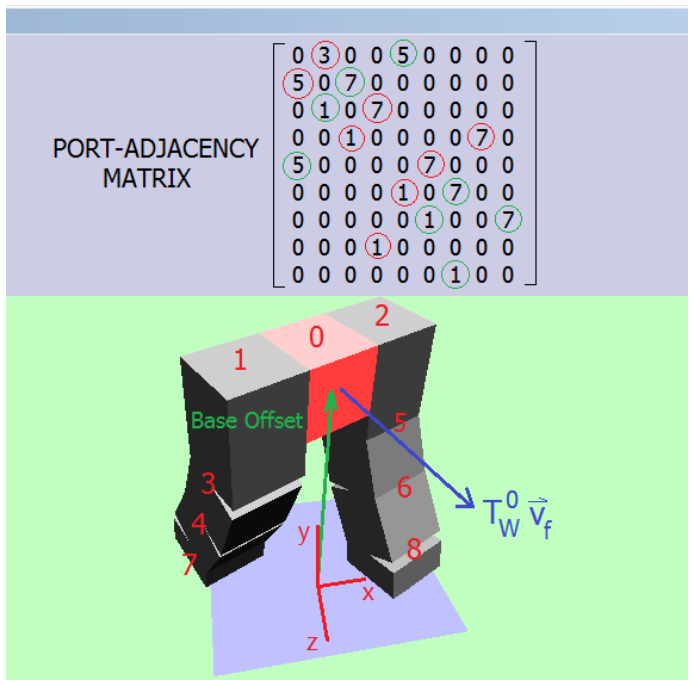

Figure 2.3: A simulated CKBot configuration with port-adjacency matrix. T_W^0 is the **base transformation matrix** dictating what translation and rotation the robot is spawned with, and $\vec{v_f}$ is the **forward vector** used to keep track of the orientation of the robot on the 2D ground plane.

Modular robot configurations can locomote or manipulate by commanding each individual module. A **gait** is the repeated actuation of every module in a configuration. There are two ways to represent gaits: *Periodic* and *Fixed* gaits.

A **periodic gait** denotes repeated motion of a configuration using sinusoids. Equa-

tion (2.1) describes the angular motion θ_i of each module, where θ_i is the angle of module $i \in \{0, 1, \ldots, n-1\}$ for an n-module configuration. Due to the simple motion laws associated with sinusoids, only 3 parameters are necessary for each module: *Amplitude* A_i, *Frequency* ω_i and *Phase* ϕ_i.

$$\theta_i = A_i \sin(\omega_i t + \phi_i) \tag{2.1}$$

A **fixed gait** is a set of reference joint angles and an associated gait execution time t_g that describes how long the robot should take to complete one gait iteration. Fixed gaits allow for more general motion of a modular robot since there is no restriction on sinusoidal behavior. However, a fixed gait representation requires more memory for storage and computational power for execution. Fixed gaits are used in the form of a **Gait Control Table (GCT)** in [34].

$$GCT = \begin{bmatrix} \theta_{11} & \theta_{21} & \ldots & \theta_{n1} \\ \theta_{12} & \theta_{22} & \ldots & \theta_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{1m} & \theta_{2m} & \ldots & \theta_{nm} \end{bmatrix} \tag{2.2}$$

Equation (2.2) above shows an implementation of a GCT in matrix form. For m gait steps, θ_{ij} corresponds to the jth reference angle command for module i. These gait steps are then linearly interpolated at every sampled time in execution such that one iteration of the gait is completed in time t_g.

2.4.3 High-Level Control - LTLMoP

The problem outlined in Section 2.3 extends the work in [21, 20, 11], where high-level control of a non-reconfigurable mobile robot is performed through the following:

1. A **discrete abstraction** of the robot's motion, sensors and actuators. The workspace is represented as a graph where each node describes state of the problem. Namely, each node denotes a region (in the set R) and the sensor information and actions the robot can perform (in the sets X and A, respectively) abstracted into binary propositions.

2. The required **task specification** S described using a subset of LTL known as GR(1) [29] or using the structured English grammar which is then automatically parsed into LTL formulas.

3. An **automaton** satisfying the LTL formula that is synthesized if the task specification is possible to satisfy. More information about the automaton and guarantees on correctness can be found in Section 2.5.2.

4. A **continuous-time execution** of the automaton. The robot has a set of low-level controllers that continuously implement discrete transitions in the automaton. For example, moving from region $R1$ to region $R2$ is a discrete transition in the automaton, but in continuous time involves motion planning from the robot's current position inside $R1$ through a transition facet between $R1$ and $R2$.

If the low-level controllers described above can appropriately carry out these transitions in the discrete abstraction of the problem, the hybrid controller generated guarantees that the robot satisfies S. Section 2.5 shows how the task space is enriched for reconfigurable modular robots with the set of traits T such that the augmented problem also has guarantees of correctness.

The approach in this paper uses the Linear Temporal Logic Mission Planning (LTLMoP [11]) framework. LTLMoP is a Python toolkit that allows a user to control a simulated or physical robot from a task specification written in structured English or LTL.

It allows the user to define a task specification, draw a workspace consisting of regions, generate the automaton satisfying the specification and actuate the robot. LTLMoP has been used with a variety of platforms such as Pioneer (as in [11]), as well as legged and humanoid robots.

2.5 Approach

2.5.1 Traits

A **trait** T_i is a high-level descriptor for the properties of a modular robot. It is mapped to a (possibly empty) set of configuration-gait pairs. A **trait mapping** $\Gamma : T \to 2^G$ is defined such that $\Gamma(T_i)$ is the set of configuration-gait pairs that corresponds to a trait $T_i \in T$. Γ can be defined manually by the user through two different techniques (outlined in Section 2.6). Classifying configuration-gait pairs $g \in G$ with traits is necessary, as automated selection of g requires knowledge of which traits correspond to which configuration-gait pairs.

The mapping Γ is captured in a **Configuration-Gait-Trait Library**. Each entry in the library corresponds to a trait T_i and its associated configuration-gait pairs $\Gamma(T_i)$. Whenever a new configuration-gait pair g is created, there is a user defined **inverse trait mapping** $\Gamma^{-1} : G \to 2^T$ such that $\Gamma^{-1}(g)$ is the set of all traits used to describe the configuration-gait pair g. The inverse trait mapping is converted to the trait mapping as follows: for each trait $T_i \in \Gamma^{-1}(g)$, $\Gamma(T_i)$ is now assigned the additional member g. Table 2.1 shows some example traits and corresponding configuration-gait pairs.

Suppose that a new configuration-gait pair "Hexapod.run" with the traits "large" and "legged" is added to Example 1 (in Section 2.3). In other words, the new input to the configuration-gait-trait library is $\Gamma^{-1}(\text{Hexapod.run}) = \{\text{large}, \text{legged}\}$. The new

Table 2.1: Sample List of Traits and Configuration-Gait Pairs

Traits	**Configuration-Gait Pairs**
Fast	Hexapod.run, Loop.roll, FoldOver.slink
Nonholonomic_Turning	Tripod.crawl, Tee.crawl, Snake.crawl, Hexapod.run
Low	Tripod.crawl, Tee.crawl, Snake.crawl
Stationary	Cross.foldup, Biped.splits, TeeStationary.swim
Large	Hexapod.run
Legged	Tripod.crawl, Hexapod.run, Biped.splits
1D_Motion	Loop.roll, FoldOver.slink
Narrow	Snake.crawl, Loop.roll, FoldOver.slink

relevant trait mappings and sets for this problem are:

- $\Gamma(\text{legged}) = \{\text{Tripod.crawl}, \text{Biped.splits}, \text{Hexapod.run}\}$
- $\Gamma(\text{narrow}) = \{\text{Loop.roll}, \text{Snake.crawl}\}$
- $\Gamma(\text{large}) = \{\text{Hexapod.run}\}$
- $G = \Gamma(\text{legged}) \cup \Gamma(\text{narrow}) \cup \Gamma(\text{large})$

Generation of a hybrid controller to solve a specification consists of 1) generating a discrete automaton and 2) executing the automaton in continuous time through low-level controllers. The control dictated by the automaton must be checked for correctness by ensuring that every possible set of active traits in the states of the automaton have a non-empty mapping (refer to 2.5.2 for more information). Also, the gaits chosen by the configuration-gait-trait library according to traits in the automaton must be executed in continuous time to drive a robot in the workspace. When no traits are required, the modular robot will use its **default configuration-gait pair** $g_{default}$.

2.5.2 Guaranteeing Correct Control

For a given task, assuming it is realizable and satisfiable (that is, there are no contradictions or impossible requirements), automaton $\mathscr{A} = (\{X\}, \{A, R, T\}, Q, Q_0, \gamma, \mathscr{T}, \delta)$ is synthesized such that:

- X is the set of **environment propositions** (or sensor information),
- $\{A, R, T\}$ is the set of **robot propositions** (actions, regions and traits),
- $Q \subset \mathbb{N}$ is the set of **states**,
- $Q_0 \in Q$ is the set of **initial states**,
- $\gamma : Q \to 2^{\{A,R,T\}}$ is the **state labeling function** where $\gamma(q) \subseteq \{A, R, T\}$ is the set of robot propositions that are true in state q,
- $\mathscr{T}(q) \subseteq T, \mathscr{T}(q) = \gamma(q) \cap T$, is the set of **active traits**, or traits that are true in state q, and
- $\delta : Q \times 2^X \to Q$ is the **transition relation**, i.e., $\delta(q, \mathscr{X}) = q' \in Q$ where $q \in Q$ is a state and $\mathscr{X} \subseteq X$ is the subset of sensor propositions that are true.

The modular robot is guaranteed to satisfy S only if it can execute $\mathscr{A}$. For correctness, the controller synthesis algorithm must ensure that all possible trait combinations $\mathscr{T}(q)$ for all the states q in the automaton have at least one configuration-gait pair corresponding to them. A task specification S is guaranteed to be correct if and only if $\Gamma(\mathscr{T}(q)) \neq \emptyset, \forall q \in Q$. Algorithm 1 shows how this check is performed.

The discrete automaton is executed in continuous time by calling basic motion planning controllers that dictate how the robot must move between regions. Based on these instructions, the best gait (e.g. a left-turning gait instead of a forward gait) in

Algorithm 1 Gait Checker

```
for q ∈ Q do
    𝒯(q) = γ(q) ∩ T
    if 𝒯(q) = ∅ then
        return g(t) = g_default
    else
        if ⋂_{τ∈𝒯(q)} Γ(τ) = ∅ then
            return ERROR: Specification is not satisfiable.
        else
            return g(t) = any g ∈ ⋂_{τ∈𝒯(q)} Γ(τ)
        end if
    end if
end for
```

the current configuration-gait pair $g(t)$ is selected. Each modular robot joint angle $\theta_i(t), i = 0, \ldots, n-1$ is then prescribed by $g(t)$. It is assumed that the modular robot reconfigures instantaneously, as this approach does not currently concern the process of reconfiguration (this is a consideration for future work). That is, if the automaton has a transition $q \to q'$ with $\gamma(q) = (r_j, T_m), \gamma(q') = (r_k, T_n)$, then the controller will take the robot along a path from region r_j to r_k with joint commands in a configuration-gait pair corresponding to traits T_m . Once the robot reaches r_k it will instantly change to a configuration-gait pair corresponding to traits T_n.

The need for a gait checker can be motivated by referring to Example 1. The automaton synthesized from this specification, shown in Fig. 2.4, consists of 4 states q_1, q_2, q_3 and q_4. In order for the specification to be satisfiable, all the possible trait combinations "legged" (in q_2), "narrow" (in q_3) and "legged $\cup$ narrow" (in q_4) must have non-empty

trait mappings. In this example, there is an error because $\Gamma(\text{legged}) \cap \Gamma(\text{narrow}) = \emptyset$; state q_4 is not satisfiable given that the configuration-gait-trait library contains no configuration-gait pair that satisfies both traits.

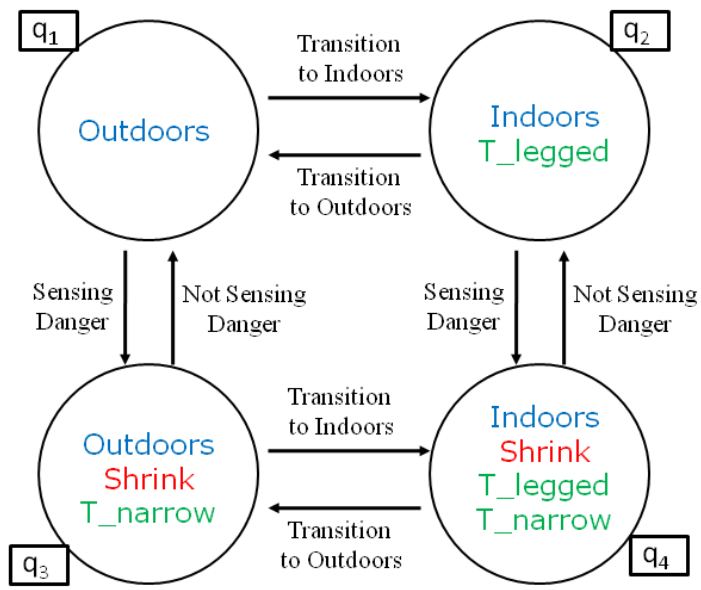

Figure 2.4: Automaton schematic for Example 1.

2.6 Populating the Library

A larger search space of configuration-gait pairs (and their associated traits) is necessary for better results. The addition of more variety in the modular robot configurations and gaits allows for more expressibility through traits; in other words, dividing a small database into "legged" versus "non-legged" does not have the discriminatory power of using many traits like "legged", "fast", "turns in place", "low", etc. However, it can be difficult to use only the definitions for configurations and gaits in Section 2.4.2 to create a collection of motion gaits for every configuration listed. The two approaches developed

to assist the user in populating the configuration-gait-trait library are as follows. The first is a manually controlled Gait Creator, and the second is a framework that uses Genetic Algorithms [12] to automatically create gaits for a modular robot configuration based on prescribed cost functions.

To assist with these tools, a 3-Dimensional physics-based simulator for modular robots was developed using the Open Dynamics Engine (ODE) [3], with CKBot as the model robot. ODE has been used in other related work such as [32]. The open-source Python PyODE bindings for ODE [4], as well as PyGame [1] and OpenGL/PyOpenGL [14, 2] for visualization enable the simulator to easily interface with LTLMoP for simulation of a reconfigurable modular robot. The simulator allows users to quickly create, test and modify different modular robot configurations and gaits.

2.6.1 Manual Population: Gait Creator

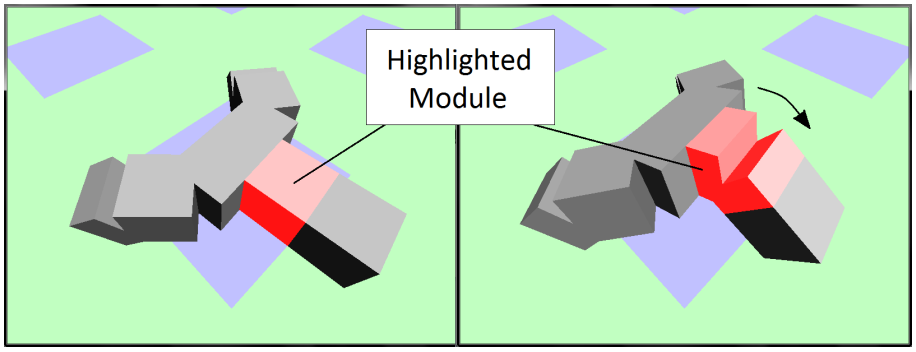

Figure 2.5: Tripod Configuration in the Gait Creator. The module being controlled is highlighted in red.

The Gait Creator allows the user to manually move each individual joint angle and see how the robot's shape changes in a physics-based environment. This way, snapshots of robot configurations can be captured and stitched together to form a gait. This gait

is written as a *fixed gait* in the same text file containing the robot's configuration information and previous gaits designed. These newly created gaits can then be executed and tested in simulation and hardware to see how well they allow the robot to move. Additionally, when creating a gait the user can choose to enter a set of traits describing the robot motion which are then automatically added to the configuration-gait-trait library. Figure 2.5 shows an example of the Gait Creator's capabilities, where the module highlighted in red is the module the user is controlling.

2.6.2 Automatic Population: Genetic Algorithms

Genetic Algorithms (GA) are useful for solving complicated optimization problems without the need to model a system. Solutions that maximize the target **fitness function** for the problem have a greater likelihood of surviving the "natural selection" process of the algorithm. A GA requires a **genotype**, which is a representation of the actual system (or **phenotype**) being modeled. For this problem, the genotype is a numerical representation of a modular robot gait (as shown in Section 2.6.2 below) and the phenotype is a simulation of the robot executing that gait. Each genotype is evaluated with the user-defined fitness function (as in Section 2.6.2. *Crossover* and *mutation* are the two common search techniques used in GAs to avoid local minima, and are explained in Section 2.6.2. Search for better solutions is performed by crossover and mutation of genotypes of probably high fitness to generate new solutions.

State Representation

Each *genotype* φ is a representation of a specific *periodic gait* (refer to Section 2.4.2) that is executed by the modular robot configuration. For a n-module configuration comprised of modules $i = 0, \ldots, n-1$, the genotype is as shown in Equation (2.3) below

such that $|\varphi| = 3n$. **Rigid modules** can also be specified in a configuration to constrain the gaits generated. This allows the user to reduce the search space of the Genetic Algorithm by deleting motion restricted modules from the optimization problem. For example, removing the "spine" of the Hexapod configuration in Figure 2.2 reduces the number of modules to search on from 25 to 18.

$$
\begin{aligned}
\varphi = X_1 Y_1 Z_1 \ldots X_n Y_n Z_n \\
\text{where } X_i \in \{0, 1, \ldots, 12, 13\} \\
\text{s.t. } A_i = 5X_i \\
\text{and } Y_i \in \{0, 1, \ldots, 8, 9\} \\
\text{s.t. } \omega_i = Y_i \\
\text{and } Z_i \in \{0, 1, \ldots, 6, 7\} \\
\text{s.t. } \phi_i = \frac{\pi}{4} Z_i
\end{aligned}
\tag{2.3}
$$

The search space for periodic gaits has been limited in size by using an amplitude step size of 5 degrees and limit of 65 degrees (and not 90 because large amplitudes can lead to unsafe motion). Also, a frequency step size of 1 rad/s (though these values can be scaled by a multiplier if necessary) and a phase step size of $\frac{\pi}{4}$ radians (45 degrees) are used. Different resolutions or ranges of values will require different spaces for each of the free parameters X_i, Y_i and Z_i.

Traits and Fitness Functions

Referring to Section 2.5, a trait $T_i \in T$ is a word, or set of words, that describes the motion and configuration of a modular robot. Some sample traits are "fast", "stationary" and "tall". For each trait and robot trajectory generated from a genotype φ there is a **trait**

score $J(T_i, \varphi)$ that the genetic algorithm is maximizing. At the start of every instance of the GA, the user must specify which traits to evaluate on. The fitness functions for all the prescribed traits are then multiplied together to give the fitness function $f(T, \varphi)$ as shown in equation (2.4).

$$f(T, \varphi) = \prod_{T_i \in T} J(T_i, \varphi) \tag{2.4}$$

The trait score $J(T_i, \varphi)$ is defined differently for every trait T_i. Note that the trajectory of φ contains all the relevant trajectory information $\{x, y, \theta, z\}$, where $x = x(t), y = y(t), \theta = \theta(t)$ is the 2D trajectory along the ground (x, y) plane and $z = z(t)$ is the robot height away from the ground. By default, the algorithm assumes that the trajectory is that of a single *base module* that depicts the center position of the robot. Other user-defined trait costs may require the positions of specific modules in the configuration. Some example trait scores are shown below.

Fast: The robot moves a large distance away from its origin

$J(\text{Fast}, \varphi) = \sqrt{(y_{final} - y_{initial})^2 + (x_{final} - x_{initial})^2}$

Stationary: The robot remains close to its origin

$J(\text{Stationary}, \varphi) = \frac{1}{J(\text{Fast}, \varphi)}$

Forward: The robot moves in the positive x direction but

deviates little in the y direction

$J(\text{Forward}, \varphi) = \frac{J(\text{Fast}, \varphi)}{\max(\max(y), |\min(y)|)}$

TurnLeft: The robot turns left regardless of translation

$J(\text{TurnLeft}, \varphi) = \text{mean}(\theta_{i+\Delta t} - \theta_i)$

for time indices $i = 0,\ \Delta t,\ 2\Delta t, \ldots,\ \lfloor \frac{t_{final}}{\Delta t} \rfloor \Delta t$ and Δt of choice

TurnInPlaceLeft: The robot turns left with little translation

$J(\text{TurnInPlaceLeft}, \varphi) = \frac{J(\text{TurnLeft}, \varphi)}{J(\text{Fast}, \varphi)}$

1DMotion: The robot has very little angular deflection

$J(\text{1DMotion}, \varphi) = \frac{1}{\text{std}(\theta)}$

Tall: The base module of the robot is far from the ground

$J(\text{Tall}, \varphi) = \text{mean}(z)$

Selection, Crossover and Mutation

The first generation of gait genotypes is instantiated randomly. After that, members of the next generation are selected based on the fitness functions through **roulette selection**. That is, the probability of a genotype $\varphi_j \in \Phi$ to be selected for the following generation is directly proportional to $f(T, \varphi_j)$, where Φ is the set of all genotypes of the previous generation, $|\Phi| = p$. A negative fitness function may arise from negative trait scores; for example, if the robot was instructed to turn left but a particular gait causes the robot to turn right, this genotype will be scored negatively. To account for negative trait scores in the roulette selection process, the minimum score (if negative) is subtracted from every score, thus ensuring there are no negative weights assigned.

Once a member has been selected, there is a chance of *crossover*, or combination between two genotypes. With probability P_{CROSS}, another genotype φ_k, $k \neq j$ will

be selected from the previous generation. Then, a **crossover index** c and one of two *crossover directions* are randomly selected, as shown below for a n-module genotype (of total length $3n$). In other words, the new genotype is either a combination of φ_j first and φ_k second, or vice-versa.

The two genotypes are:

$$\varphi_j = j_1 j_2 j_3 \ldots j_{3n-2} j_{3n-1} j_{3n}$$
$$\varphi_k = k_1 k_2 k_3 \ldots k_{3n-2} k_{3n-1} k_{3n}$$

The two possible crossovers are:

DIRECTION 1: $\varphi_{new} = j_1 j_2 \ldots j_{c-1} k_c \ldots k_{3n-1} k_{3n}$

DIRECTION 2: $\varphi_{new} = k_1 k_2 \ldots k_{c-1} j_c \ldots j_{3n-1} j_{3n}$

Note that each element of a genotype corresponds to either an element X, Y or Z, within the respective domains as in section 2.6.2. The equivalence is as follows: $\varphi_j = j_1 j_2 j_3 \ldots j_{3n-2} j_{3n-1} j_{3n} = X_1 Y_1 Z_1 \ldots X_n Y_n Z_n$.

Assume that after crossover, the new genotype φ_{new} is now

$$\varphi_{new} = n_1 n_2 n_3 \ldots n_{3n-2} n_{3n-1} n_{3n}$$

Finally, with probability P_{MUTATE}, a **mutation index** μ is chosen such that the element n_μ is changed randomly to another legal value. Again, this depends on whether n_μ is an X, Y or Z value corresponding to amplitude, frequency or phrase (respectively).

Limitations

There are some complications with the GA framework. Firstly, solutions are not neccesarily close to optimal and convergence of the algorithm towards a "good" solution is

not guaranteed to occur at any particular time. Secondly, users can specify traits to synthesize contradictory fitness functions. A simple example is using the traits "fast" and "stationary", as their trait scores are reciprocals of each other and the fitness function would equate to 1 for any non-zero displacement of the robot. Composing a fitness function with trait scores requires judicious human selection of traits. Future work involves investigation of automatically selecting trait scores. Lastly, there are many parameters (simulation time/step size, crossover/mutation rates, population sizes, termination conditions) to this algorithm. The "best" values to choose vary with every aspect of the problem, ranging from the traits being used to how many modules a configuration consists of. A MATLAB post-processing script is used to generate graphs for validation of results (refer to Figures 2.7 and 2.8).

2.7 Results

2.7.1 Gait Creator

A sample gait created manually using the Gait Creator is shown in Figure 2.6. This "Slinky.slink" configuration-gait pair causes the robot to move forward by folding over itself. The Gait Creator was useful in creating such a gait that required fine tuning to avoid falling in an incorrect direction. This configuration-gait pair was assigned with the traits *narrow*, *1D_motion* and *handles_steps*. The "Hexapod.run" configuration-gait pair, pictured in Figure 2.2, was inspired by motion of insects such as cockroaches [28]. This 25-module robot and its motion were given the traits *large*, *legged* and *nonholonomic_turning*.

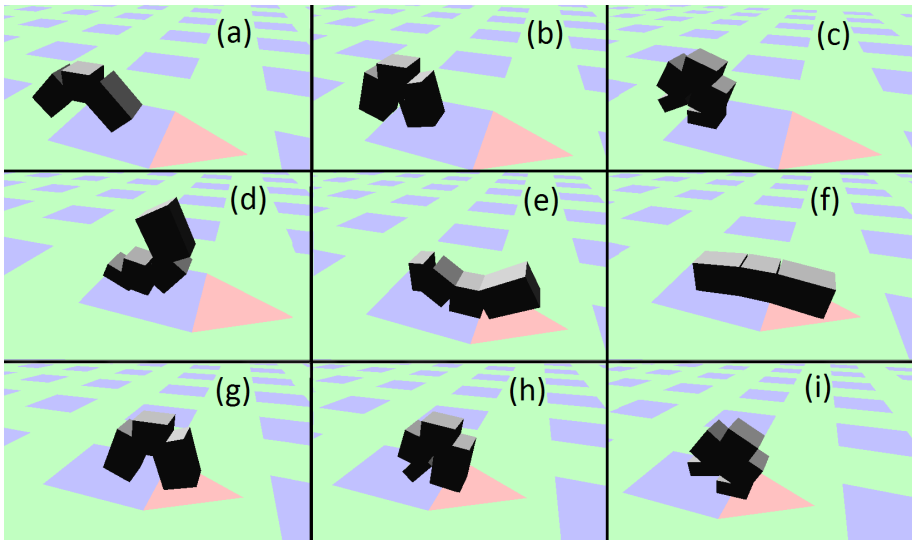

Figure 2.6: Manually generated Slinky gait.

2.7.2 Genetic Algorithms

The examples in Figures 2.7 and 2.8 have been performed with 50 generations and a population size $p = 50$. All the examples use probabilities $P_{CROSS} = 0.5$ and $P_{MUTATE} = 0.15$, and each genotype is evaluated over 350 simulation timesteps amounting to 11.7 seconds of motion.

In Figure 2.7, a turning gait is generated for the 9-module *Cross* configuration under the trait *TurnInPlaceLeft*. The algorithm is therefore scored on how little the base module moves from the starting position, and how quickly it is able to turn in a clockwise direction. In Figure 2.8, the traits *Tall* and *TurnRight* are prescribed for the 4-module *Snake* configuration. The simulation screenshots show that the base module (in this case, the "tail" module) is lifted off the ground as the robot turns to the right. The units of the graph are normalized to the module height. The center of the *Snake* configuration in its rest position is at a height of 0.5 modules and peaks at approximately 1.5 modules tall.

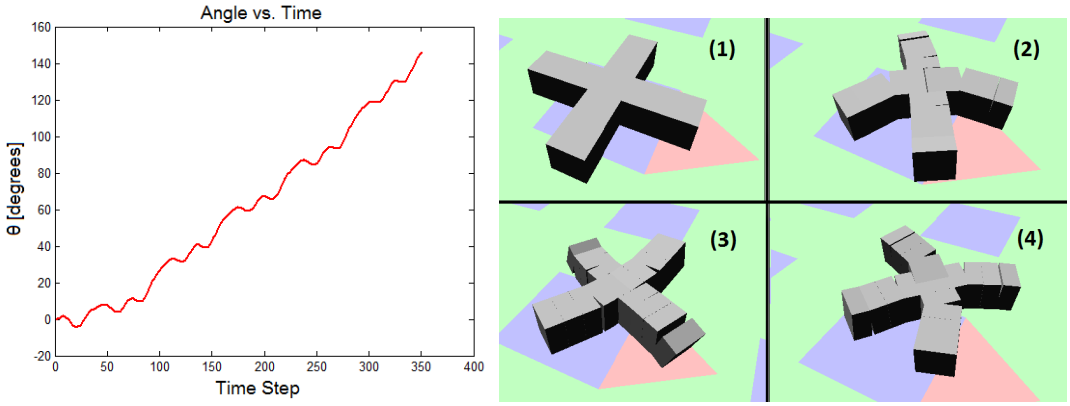

Figure 2.7: Best Orientation time history for Cross configuration under trait "TurnInPlaceLeft" (Top). Screenshots of configuration running the resulting gait (Lower).

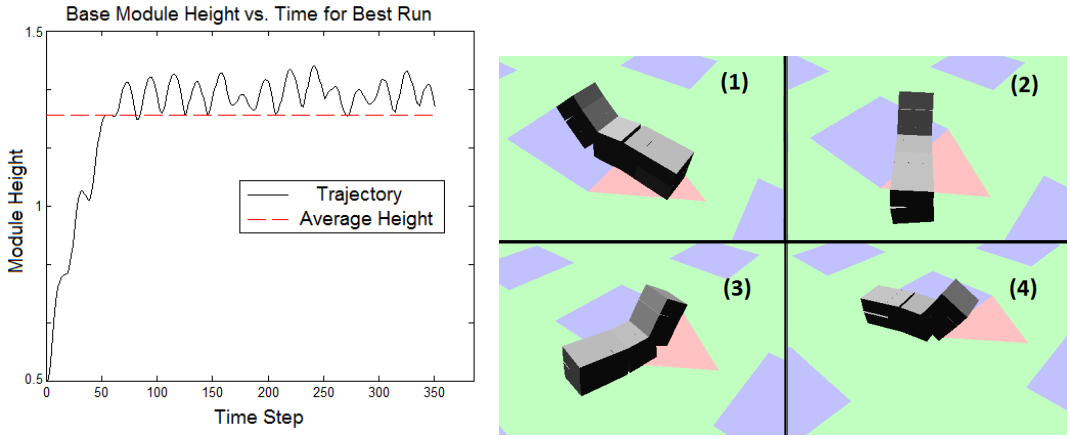

Figure 2.8: Best base module height time history for Snake configuration under the traits "Tall" and "TurnRight"(Top). Screenshots of configuration running the resulting gait (Lower).

2.8 Example: Rescue Robot

2.8.1 Task Specification

```
### Initial conditions and safety. ###
Robot starts in Safehouse
Env starts with false
Always not Mountains

### Rescue mission goals. ###
If you are sensing distress_signal then
  visit Rescue_Point
carrying_person is set on Rescue_Point and
  reset on Safehouse
If you are activating carrying_person or you
  activated carrying_person then visit Safehouse
If you are activating carrying_person or
  you activated carrying_person then do
  not Trench
Do not Trail unless you activated carrying_person
  or you are activating carrying_person
If you are not sensing distress_signal and
  you are not activating carrying_person then
  visit Watchtower

### Reconfiguration goals. ###
Do T_narrow if and only if you were in
  between LeftOfTrench and RightOfTrench
Do taking_cover if and only if you are sensing
  air_raid
Do T_low and T_legged if and only if you are
  activating taking_cover
```

Figure 2.9: Task Specification in structured English.

The capabilities of the approach are demonstrated through a rescue robot scenario executed in simulation and with a real robot. The high-level task specification in structured

English is as shown in Figure 2.9 and the workspace for this example is shown in Figure 2.10. The lines labeled with a "###" are comments, and therefore not part of the specification.

The robot must begin in the *Safehouse* region and stay away from the *Mountains* region. It is commanded to visit the *Watchtower* and remain there to patrol unless a *distress signal* is sensed. In this case, the robot must pick up a person from the *Rescue Point* and return to the *Safehouse* with the person. When *carrying a person*, the *Trench* is too dangerous so it is unavailable. The task specification also has requirements on the configuration-gait pair of the robot by using the trait prepositions *T_narrow*, *T_low* and *T_legged*. If the robot is in between the regions bordering the *Trench* (that is, inside the *Trench*), it must adopt the trait *T_narrow* in order to fit in this region. If the robot senses an *air raid*, it must perform the action *taking cover*. When the robot is *taking cover*, it will adopt the traits *T_low* and *T_legged*.

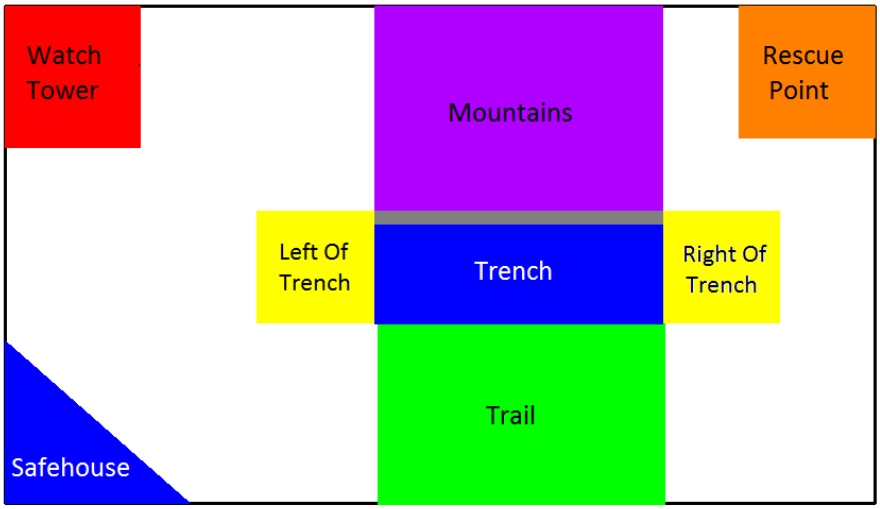

Figure 2.10: Workspace for the task specification.

2.8.2 Execution of Task Specification

Figure 2.11 shows some screenshots of the example specification in simulation. Each image displays the simulated robot in the workspace and the simulated sensors, where the sensors highlighted with a border are active. These images show that the default configuration-gait pair used is "Snake.crawl". When the trait *T_narrow* is active, the configuration-gait-trait library selects *Loop.roll*; similarly, when the traits *T_low* and *T_legged* are active, the library selects the "Tripod.crawl" configuration-gait pair.

Figure 2.12 shows screenshots of the same task specification performed using the CKBot platform. In order to ensure that the configuration-gait-trait library picks configuration-gait pairs that are only realizable in simulation, the additional trait *T_hardware* was included. This trait, along with the structured English sentence "`Always do T_hardware`", ensures this. There are two differences with the execution in simulation. The first is that the gaits in "Tripod.crawl" for the hardware are different; in the simulation example, they were created using the Gait Creator and on CKBot they were automatically synthesized using the GA framework. The second difference is that the library now selects the "Snake.crawl" configuration-gait pair instead of "Loop.roll" when the trait *T_narrow* is active because "Loop.roll" is not labeled with *T_hardware*.

2.9 Conclusions and Future Work

This paper detailed an approach for provably correct control of reconfigurable modular robots through a configuration-gait-trait library. This has been demonstrated both in simulation and with the CKBot hardware. Two methods (one manual and one automatic) were presented to facilitate the population of this library through gait creation. The novelty in this approach is the dual use of *traits* to describe modular robots; traits are

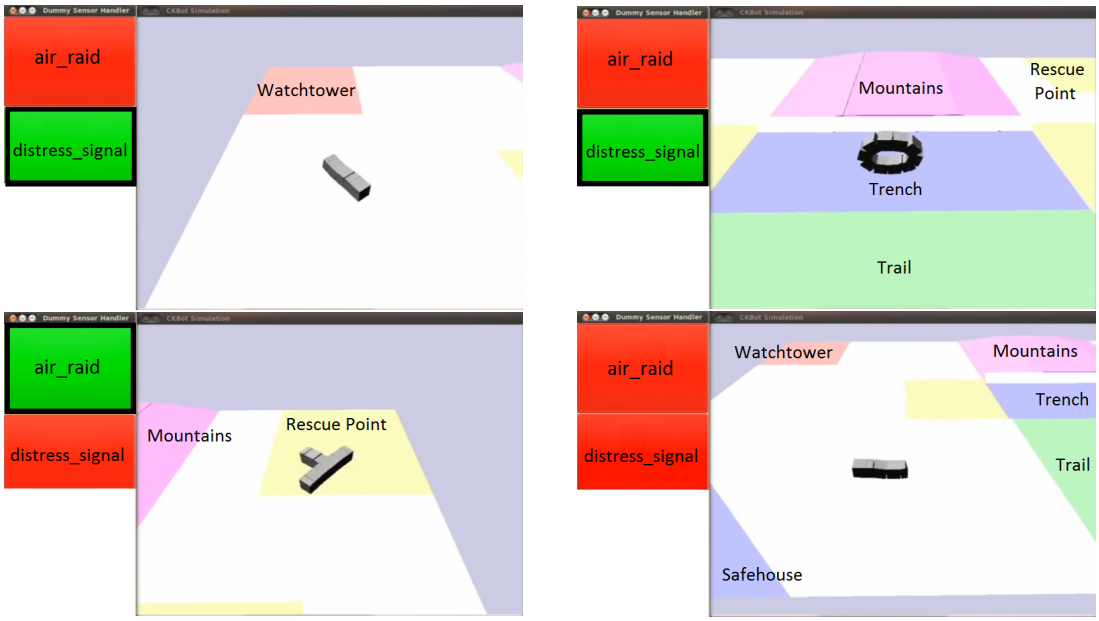

Figure 2.11: Simulation screen shots. The images, from top to bottom, left to right, show: a) The default Snake.crawl configuration-gait pair moving in the environment, b) The Loop.roll configuration-gait pair in the Trench, c) The Tripod.crawl configuration-gait pair when air_raid is active and d) Snake.crawl returning to the Safehouse while carrying_person is true.

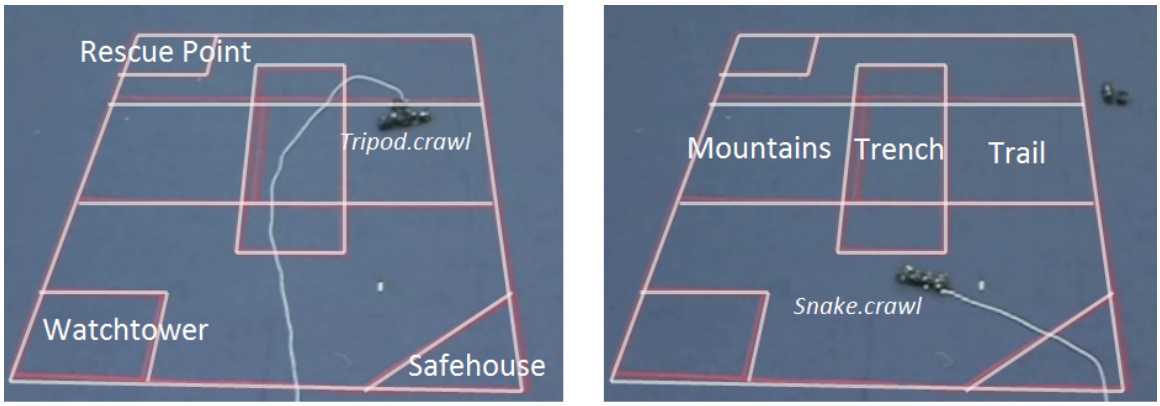

Figure 2.12: Hardware screen shots showing CKBot powered via Ethernet, with Vicon markers for pose information. The Tripod.crawl configuration-gait pair moving in the Trail region as air_raid is being sensed (left) and the default Snake.crawl configuration-gait pair (right).

used as binary propositions in the high-level control problem, and also to synthesize fitness functions for the Genetic Algorithm gait generator.

Future work directions include: (a) Adding reconfiguration controllers to replace instantaneous robot transformations, (b) Experimenting with additional modular robot platforms and (c) Improving the Genetic Algorithm gait generator by exploring different options. Some examples of alterations to be investigated are: (i) using gait types that are not uncoupled sinusoids (such as CPGs [24, 16] or Gait Control Tables [34]), (ii) extending the set of traits to deal with more complicated or specialized types of motion, (iii) exploring higher-level ways to express fitness functions through natural or structured language and (iv) optimizing robot geometries as well as gaits through reconfiguration (as is done in [25]).

2.10 Acknowledgments

We thank Jimmy Sastra and Dr. Mark Yim for providing us with the CKBot hardware and software, and providing support. We also thank Sarah Koehler for her contributions to this research.

BIBLIOGRAPHY

[1] Pygame wiki. http://www.pygame.org/wiki/about.

[2] Pyopengl 3.x, the python opengl binding. http://pyopengl.sourceforge.net/.

[3] Open dynamics engine (ode) community wiki. http://opende.sourceforge.net/wiki/index.php/Main_Page, April 2010.

[4] Pyode. http://pyode.sourceforge.net/, 2010.

[5] Robots using ros: Mini-pr2. http://www.ros.org/news/2010/08/robots-using-ros-mini-pr2.html, August 2010.

[6] J. Ahuactzin, E. Talbi, P. Bessire, and E. Mazer. Using genetic algorithms for robot motion planning. In Christian Laugier, editor, *Geometric Reasoning for Perception and Action*, volume 708 of *Lecture Notes in Computer Science*, pages 84–93. Springer Berlin / Heidelberg, 1993.

[7] A. Bhatia, L.E. Kavraki, and M.Y. Vardi. Sampling-based motion planning with temporal goals. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2689–2696, Anchorage, Alaska, May 3 2010.

[8] S. Castro, S. Koehler, and H. Kress-Gazit. High-level Control of Modular Robots. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on, to appear*, September 2011.

[9] C. Chiang and G. Chirikjian. Modular robot motion planning using similarity metrics. *Autonomous Robots*, 10:91–106, 2001. 10.1023/A:1026552720914.

[10] E. A. Emerson. Temporal and modal logic. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[11] C. Finucane, G. Jing, and H. Kress-Gazit. LTLMoP: Experimenting with language, temporal logic and robot control. In *IEEE/RSJ Int'l. Conf. on Intelligent Robots and Systems*, pages 1988 – 1993, Taipei, Taiwan, October 2010.

[12] S Forrest. Genetic algorithms: principles of natural selection applied to computation. *Science*, 261(5123):872–878, 1993.

[13] S. Grillner and P. Wallen. Central pattern generators for locomotion, with special reference to vertebrates. *Annual Review of Neuroscience*, 8(1):233–261, 1985.

[14] Khronos Group. Opengl api documentation overview. http://www.opengl.org/documentation/, 2011.

[15] B. Johnson and H. Kress-Gazit. Probabilistic analysis of correctness of high-level robot behavior with sensor error. In *Proceedings of Robotics: Science and Systems*, Los Angeles, CA, USA, June 2011.

[16] A. Kamimura, H. Kurokawa, E. Yoshida, S. Murata, K. Tomita, and S. Kokaji. Automatic locomotion design and experiments for a modular robotic system. *Mechatronics, IEEE/ASME Transactions on*, 10(3):314 –325, june 2005.

[17] A. Kamimura, S. Murata, E. Yoshida, H. Kurokawa, K. Tomita, and S. Kokaji. Self-reconfigurable modular robot - experiments on reconfiguration and locomotion. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 1, pages 606–612, 2001.

[18] S. Karaman and E. Frazzoli. Complex mission optimization for multiple-uavs using linear temporal logic. In *American Control Conference*, Seattle, Washington, 2008.

[19] M. Kloetzer and C. Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transaction on Automatic Control*, 53(1):287–297, 2008.

[20] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Translating structured english to robot controllers. *Advanced Robotics Special Issue on Selected Papers from IROS 2007*, 22(12):13431359, 2008.

[21] H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas. Temporal-logic-based reactive mission and motion planning. *Robotics, IEEE Transactions on*, 25(6):1370 –1381, dec. 2009.

[22] H. Kress-Gazit and G.J. Pappas. Automatic synthesis of robot controllers for tasks with locative prepositions. In *IEEE International Conference on Robotics and Automation*, pages 3215–3220, Anchorage, Alaska, 2010.

[23] M. Lahijanian, J. Wasniewski, S.B. Andersson, and C. Belta. Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3227 –3232, may 2010.

[24] D. Marbach and A.J. Ijspeert. Co-evolution of Configuration and Control for Homogenous Modular Robots. In F. Groen, editor, *Proceedings of the Eighth Conference on Intelligent Autonomous Systems (IAS8)*, pages 712–719. IOS Press, 2004.

[25] D. Marbach and A.J. Ijspeert. Online optimization of modular robot locomotion. In *Mechatronics and Automation, 2005 IEEE International Conference*, volume 1, pages 248 – 253, July 2005.

[26] A. Pamecha, I. Ebert-Uphoff, and G.S. Chirikjian. Useful metrics for modular robot motion planning. *Robotics and Automation, IEEE Transactions on*, 13(4):531 –545, aug 1997.

[27] M. Park, S. Chitta, A. Teichman, and M. Yim. Automatic configuration recognition methods in modular robots. *Intl J. of Robotics Research (invited)*, November 2006.

[28] F. Pfeiffer, J. Eltze, and H. Weidemann. Six-legged technical walking considering biological principles. *Robotics and Autonomous Systems*, 14(2-3):223 – 232, 1995. Research on Autonomous Mobile Robots.

[29] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. In *VMCAI*, pages 364–380, Charleston, SC, Jenuary 2006.

[30] J. Sastra, W. G. Bernal-Heredia, J. Clark, and M. Yim. A biologically-inspired dynamic legged locomotion with a modular reconfigurable robot. In *Proc. of DSCC ASME Dynamic Systems and Control Conference*, Ann Arbor, Michigan, USA, October 2008.

[31] J. Sastra, S. Chitta, and Mark Yim. Dynamic rolling for a modular loop robot. In *International Symposium on Experimental Robotics*, Rio De Janeiro, Brazil, 2006.

[32] I. Sucan, J. F. Kruse, M. Yim, and L. Kavraki. Reconfiguration for modular robots using kinodynamic motion planning. In *ASME Dynamic Systems and Control Conference*, Ann Arbor, MI, October 2008.

[33] T. Wongpiromsarn, U. Topcu, and R.M. Murray. Receding horizon control for temporal logic specifications. In *Proc. of the 13th International Conference on Hybrid Systems: Computation and Control*, 2010.

[34] M. Yim, S. Homans, and K. Roufas. Climbing with snake-like robots. In *Proc. of the IFAC Workshop on Mobile Robot Technology*, Jejudo, Korea, May 21-22 2001.

[35] M. Yim, Wei-Min Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G.S. Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *Robotics Automation Magazine, IEEE*, 14(1):43 –52, march 2007.

[36] M. Yim, P.J. White, M. Park, and J. Sastra. Modular self-reconfigurable robots. In *Encyclopedia of Complexity and Systems Science*, pages 5618–5631. 2009.

[37] M. Yim, Ying Zhang, and D. Duff. Modular robots. *Spectrum, IEEE*, 39(2):30 –34, feb 2002.

[38] E. Yoshida, S. Murata, A. Kamimura, K. Tomita, H. Kurokawa, and S. Kokaji. Motion planning for a self-reconfigurable modular robot. In *Experimental Robotics VII*, volume 271 of *Lecture Notes in Control and Information Sciences*, pages 385–394. Springer Berlin / Heidelberg, 2001.