High-Level Control of Modular Robots

Sebastian Castro, Sarah Koehler, Hadas Kress-Gazit
Sibley School of Mechanical and Aerospace Engineering
Cornell University, Ithaca NY
{sac77, smk269, hadaskg}@cornell.edu

***Abstract*— This paper discusses the creation of provably correct control for modular robots from high-level tasks expressed using sentences in structured English. Due to the nature of modular robots, we address problems that include requirements on the geometry and motion characteristics of the robot; these requirements are captured using *traits* in the specification that are then used in the control generation process.**

Outlined in this paper is our approach for generating all the lower levels of control for a modular robot given the high-level problem statement. The approach includes the use of a configuration-gait-trait library for characterizing modular robots and tools for populating this library such as a physics-based simulator and gait creator. The approach is demonstrated in simulation and with the CKBot hardware platform.

I. Introduction

One of the main challenges in robotics is programming robots to perform complex and interesting tasks while at the same time guaranteeing their behavior is safe and predictable. Recently, there has been work on generating correct-by-construction control for mobile robots from high-level specifications expressed using temporal logic (see, for example [1], [2], [3], [4], [5]) or structured language (as in [6]). There, the tasks usually include a specification regarding the motion of the robot in the environment, and for some of the approaches ([1], [5]), reactions to information obtained from the environment through sensors.

In this paper we consider high-level tasks for chain-type modular robots [7], [8]. Modular robots are unique because the variety of configurations and gaits they can exhibit allows for a richer set of tasks; one can specify not only motion in the environment and reaction to environmental events, but also utilize *traits* as descriptors of the motion and morphology. For example, the robot can be required to have a "narrow" configuration when traveling down a corridor and a "low" motion profile when moving in an area with a low ceiling such as under a table. In this paper we address the problem of generating provably correct control for modular robots from high-level specifications given in structured English and containing traits that describe the desired qualities of the motion.

II. Problem Formulation

In order to automatically transform high-level specifications, given in structured English, into provably correct low-level control of modular robots, we build on the work in [1], [6], [9]. However, we also specifically address modular robots by enriching the task space with *traits* as additional requirements on morphology and gaits.

We consider a modular robot comprised of a set of connected modules moving around an environment P. The robot's continuous trajectory is described as $p(t) \in P, \forall t > 0$. The robot is capable of performing different actions such as turning on a camera or sounding an alarm. Furthermore, we define the configuration-gait pair $g(t) \in G$ of a robot, where G is the set of all configuration-gait pairs. $g(t)$ describes the morphology of the robot and the type of motion gaits being used at time t.

The required robot behavior is captured using structured English sentences belonging to the grammar described in [6], [10] where the basic lexicon consists of the following:

- Set of sensor names X corresponding to sensor information the robot can obtain. We assume these are binary sensors; that is, they can either be true or false.
- Set of region names R corresponding to regions of interest in the workspace.
- Set of binary actions A that the robot can perform. During robot execution, we denote the set of active actions (actions currently being executed) as $a(t) \in 2^A$ where 2^A denotes the power set of A.
- Set of traits T corresponding to motion types the robot can exhibit. We distinguish traits by adding a prefix "T_" to each trait. For example, T_low represents configurations and gaits that cause the robot to stay close to the ground. As discussed in Section IV, we define the mapping $\Gamma : T \rightarrow 2^G$ such that $\Gamma(T_i)$ is the set of configuration-gait pairs that is labeled with trait T_i.

A specification S is a set of English sentences belonging to a predefined grammar and the above lexicon. The sentences, restricted to a subset of Linear Temporal Logic (LTL [11]), can capture conditional statements (e.g. "If you are sensing predator then visit safePlace"), safety requirements (e.g. "Always not unsafeRegion"), non-projective locative prepositions such as "between" and more. We refer the reader to [6], [10] for a description of the grammar.

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

[1]As discussed in [1], the specifications may include assumptions about the behavior of the environment, for example "a predator cannot appear in region Tunnel". An admissible environment is one that satisfied all the assumptions.

III. Background

A. High-Level Control for Non-Reconfigurable Robots

The high-level control approach outlined in Section II can be summarized by the following steps:

1) A discrete abstraction of the robot's motion, sensors and actuators is created by representing the workspace as a graph where each node corresponds to a region (the set R as mentioned in Section II) and by abstracting the sensor information the robot can collect at runtime and actions the robot can perform into binary propositions (the sets X and A respectively).
2) The required task is described using a subset of LTL known as GR(1) [12] or using the structured English grammar that is then automatically parsed into LTL formulas.
3) An automaton satisfying the LTL formula is synthesized if the task can be done. Section IV-B provides more details regarding the automaton.
4) The automaton is continuously executed by the robot by calling basic controllers that continuously implement every discrete transition in the automaton. For example, moving from a state where the proposition $room_1$ is true to a state where $room_2$ is true will require calling a controller capable of driving the robot from room 1 to room 2.

Given an appropriate set of basic controllers, the resulting hybrid controller is guaranteed to drive the robot such that it achieves its task. In the next sections, we describe how the task space and the resulting control can be enriched to include traits and corresponding configurations and gaits.

The Linear Temporal Logic Mission Planning (LTLMoP [9]) framework is a Python toolkit that allows a user to control a simulated or physical robot from structured English instructions. It allows the user to draw a workspace, define a task, generate the automaton satisfying the task (if the task can be guaranteed) and either simulate a robot or connect to a physical robot. LTLMoP has been shown to work with a variety of platforms including Pioneer [9] and humanoid robots. In this work, LTLMoP was enhanced to be able to control the motion, configuration and gaits of modular robots.

B. Modular Robots - Hardware

To demonstrate our approach, we experiment with the Connector Kinetic roBot (CKBot) platform, developed by Yim et. al. [13] and shown in Fig. 1. Each CKBot module is cubic in shape and has a single degree of freedom in the form of a servo-actuated rotating hinge. Every module is equipped with 7 infrared receiver-transmitter pairs, or ports, distributed over 4 of the cube faces. These ports are what define every possible connection between two modules; in total, there are 40 ways to connect two CKBot units.

Prior work with CKBot has dealt with single-application locomotive gaits or with fixed-base kinematics and controls problems. Examples of this single-application locomotion with CKBot can be seen in dynamic rolling of a closed loop of modules [14] or legged gaits using compliant legs [15]. Other research has utilized CKBot for recreation of the high degree-of-freedom arms of the PR2 robot platform [16], where specialized wheel modules are used for locomotion.

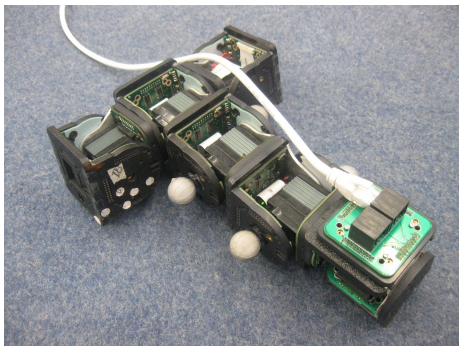

Fig. 1. Ethernet-powered CKBot configuration with markers used by the Vicon motion capture system to provide pose information.

C. Modular Robots - Configurations and Gaits

A modular robot *configuration* is a representation of the connectivity of the modules of a robot. For CKBot specifically, we use the *port-adjacency matrix* as is done in [13]. Every configuration is then assigned different ways, or *gaits*, to achieve motion. A gait can be defined as a repeatable sequential execution of joint angle commands for every CKBot module. For any modular robot, we describe its motion and geometry by defining its *configuration-gait pair*.

For our experiments we have two ways of representing gaits: *Periodic* and *Fixed* gaits. A *Periodic Gait* represents repeated motion of a configuration using sinusoids. Due to the simple motion laws associated with sinusoids, we only need to keep track of 3 parameters for each module: *Amplitude*, *Frequency* and *Phase*. For one gait, and for each module $i \in \{0,...,n-1\}$ we then assign these parameters A_i, ω_i and ϕ_i such that the equation (1) describes the angular motion θ_i of each module.

$$\theta_i = A_i \sin(\omega_i t + \phi_i) \tag{1}$$

Fixed Gaits allow for more general motion but require more memory and computational power to execute. A fixed gait is a collection of joint angle snapshots beginning and ending with the same values and an associated gait execution time t_g that describes how long the robot should take to complete one gait iteration. This structure is similar to a *Gait Control Table (GCT)* as shown in [17].

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

In equation (2) above we show a GCT as it is implemented in matrix form. For m gait steps, θ_{ij} corresponds to the j^{th} reference angle command for module i. These gait steps are then linearly interpolated at every sampled time such that the gait is executed smoothly.

IV. Approach

The control generation process can be seen as containing two phases; generation of a discrete automaton and a continuous implementation of the automaton to provide continuous control commands for the modules. For the automaton synthesis, traits are regarded as robot propositions, similar to regions and actions. Following a successful synthesis and in order to guarantee appropriate configurations and gaits exist, the automaton is checked for emptiness of traits as described below. The continuous control generation process includes choosing a configuration-gait pair according to the traits in the automaton.

A. Traits

A trait T_i is a descriptor for the motion of a modular robot that is interesting from a high-level task perspective. It is connected to a (possibly empty) set of configuration-gait pairs. We define a mapping $\Gamma : T \rightarrow 2^G$ such that $\Gamma(T_i)$ is the set of configuration-gait pairs that correspond to trait T_i. Currently Γ is defined manually by the user and we expect to automate part of that mapping in future work.

The mapping Γ is captured in a *Configuration-Gait-Trait Library*. Each entry in the library corresponds to a trait and its associated configuration-gait pairs. Whenever a configuration and associated gaits are created, the appropriate library entries are updated with the new pairs. For example, the expression 'Tripod.crawl' indicates that the robot is in the configuration called 'Tripod' and is using the 'crawl' gaits. This pair is classified with the traits 'low', 'nonholonomic_turning' and 'legged'. Table I shows some example traits and corresponding configuration-gait pairs.

TABLE I
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

B. Guaranteeing Correct Control

For a given task, assuming it is feasible (that is, there are no contradictions or impossible requirements), the synthesis algorithm generates an automaton $\mathscr{A} = (\{X\},\{R,A,T\},Q,Q_0,\delta,\gamma)$ such that

- X is the set of environment propositions (sensor information),
- $\{R,A,T\}$ is the set of robot propositions (regions, actions and traits),
- $Q \subset \mathbb{N}$ is the set of states,
- $Q_0 \in Q$ is the set of initial states,
- $\delta : Q \times 2^X \rightarrow Q$ 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, and
- $\gamma : Q \rightarrow 2^{\{R,A,T\}}$ is the state labeling function where $\gamma(q) = Y$ and $Y \subseteq \{R,A,T\}$ is the set of robot propositions that are true in state q.

The modular robot is guaranteed to satisfy its task only if it can execute the generated automaton; to ensure possible execution, the control synthesis algorithm must verify that all possible required traits and trait combinations have a defined configuration-gait pair associated with them. This is done by extracting from the automaton all possible trait combinations $T^* = \cup_{q_j \in Q}\{T_i \in T | T_i \in \gamma(q_j)\}$ and checking each set of traits in T^* against the configuration-gait-trait library.

Fig. 2 shows the result of the emptiness check for two sets of specifications. On the left, the Gait Checker algorithm discovered two combinations of traits that are empty; the traits "holonomic" and "fast" are defined in the library, however there is no configuration-gait pair that is both holonomic and fast (bottom error). The combination of the two sentences in the red box result in a possible state that has the trait combination "holonomic", "fast" and "stationary" which is also empty (top error). On the right is the result of a successfully synthesized specification.

To provide the continuous control for the modular robot, the discrete automaton is executed by calling basic controllers to provide the velocity vectors the robot must follow to move from one region to the next. Then, the automaton interfaces with the configuration-gait-trait library to obtain the joint commands for the individual modules. We assume each robot has a predefined default configuration and gait so that if a state has no associated trait, the default is used. For this work we assume instantaneous reconfiguration of the robot. Thus, if the automaton has a transition $q_i \rightarrow q_j$ with $\gamma(q_i) = (r_k,T_l), \gamma(q_j) = (r_m,T_n)$ then the controller will provide joint commands that will move the robot in a configuration and gait corresponding to trait T_l along a path that leads the robot from region r_k to r_m and once the robot reaches r_m it will instantly reconfigure to a configuration and gait corresponding to trait T_n.

V. Experimental Setup

A. Modular Robots with LTLMoP

LTLMoP was designed so that communicating with a physical robot is as straightforward as it is with a simulated one. Regardless of the platform being used, the controller generated from high-level structured English specifications is the same. During execution, LTLMoP receives the robot's pose information (either from the simulation or from localization systems such as the Vicon motion capture system); in turn, LTLMoP sends the robot a gait command. Because modular robots have a finite number of gaits, this gait command is a number indicating which gait the current configuration-gait pair $g(t)$ should use. For example, if the 'Snake.crawl' pair is currently active and the motion planner needs the robot to turn left, the command sent will be to execute the left-turning gait *2*.

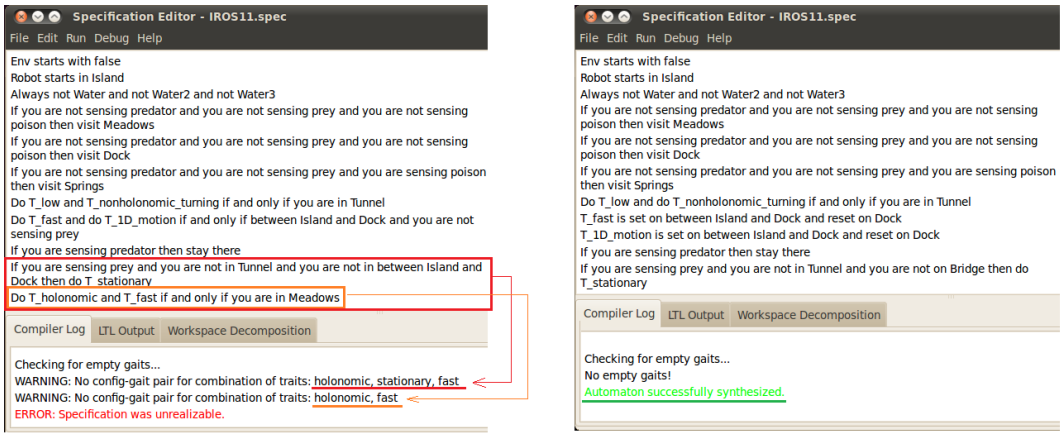

Fig. 2. Sample specifications with Gait Checker output. Unrealizable Specification (Left) and successfully synthesized specification (Right)

B. *Populating the Library*

The richness of the task space for modular robots depends on the number of traits and configuration-gait pairs. Assigning joint angles to a complicated robotic structure with the intent of motion can be quite unintuitive. In this section we describe the modular robot simulator and Gait Creator as tools that facilitate the process of populating the library with a variety of entries (traits and the corresponding configuration-gait definitions).

We developed a 3-Dimensional physics-based simulator for modular robots like CKBot. This is similar to previous work by Şucan et al. [18], where a physics engine is used to plan and simulate reconfiguration of modular robots. This simulator is built on the Open Dynamics Engine (ODE) [19], using the open-source Python bindings PyODE [20] along with PyGame [21] and OpenGL [22], [23] for visualization. This allows the simulator to easily interface with LTLMoP for simulating a modular robot. With this simulator, we can create modular robot configurations of any finite size and structure and test them without being limited by the availability of hardware.

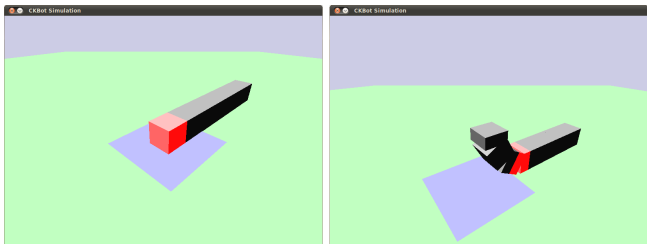

Fig. 3. FoldOver Configuration in the Gait Creator

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 3 shows an example of the Gait Creator's capabilities, where the module highlighted in red is the module the user is controlling.

VI. Executing High-Level Tasks with the CKBot Platform

We demonstrate the capabilities of modular robots in the LTLMoP framework through the following examples. The first example uses the modular robot simulator and the second uses the CKBot platform.

A. *Scenario with Simulated Robot*

The structured English sentences corresponding to these requirement written out in Structured English are shown below, for the workspace in Figure 4. The traits that are present in this example are *low, fast, nonholonomic_turning, 1D_motion* and *stationary*.

```
-Env starts with false
-Robot starts in Island
-Always not Water and not Water2 and not Water3
-If you were in Tunnel then do not sense predator or prey
-If you were in between Island and Dock then do not sense
predator or prey
-If you are not sensing predator and you are not sensing
prey and you are not sensing poison then visit Meadows
-If you are not sensing predator and you are not sensing
prey and you are not sensing poison then visit Dock
-If you are not sensing predator and you are not sensing
prey and you are sensing poison then visit Springs
-Do T_low and T_nonholonomic_turning if and only if you
are in Tunnel
-T_fast is set on between Island and Dock and reset on Dock
-T_1D_motion is set on between Island and Dock
and reset on Dock
-If you are sensing predator then stay there
-If you are sensing prey then do T_stationary
```

Starting in the *Island* region, the robot is required to visit the *Dock* or *Meadows* without ever going into the water regions. In addition, the task specification above dictates how the robot should react to the environment (i.e. sensors) and location. In the simulated execution that can be seen

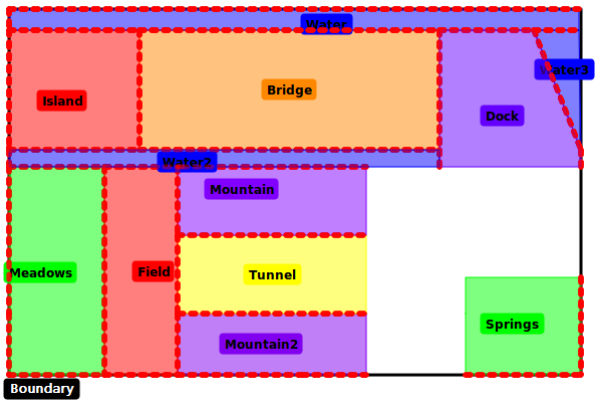

Fig. 4. Workspace for the Simulation Scenario

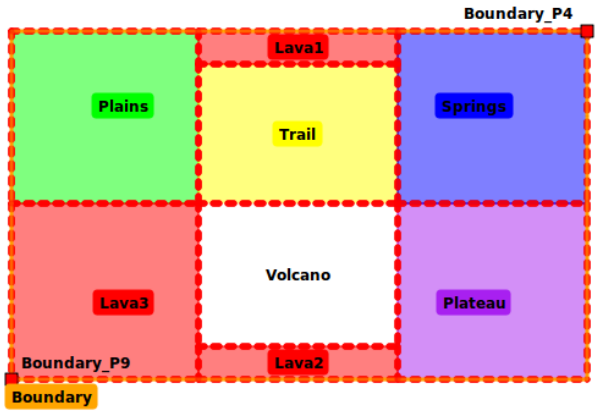

Fig. 5. Workspace for the Hardware Scenario.

in the accompanying video, the following configuration-gait pairs were automatically selected using the configuration-gait-trait library. Fig. 6 depicts the simulation environment and the sensor interface. The fast, 1D *FoldOver* configuration is used to move over the *Bridge* region. When all sensors are false and the robot is in the non-convex region, the default Hexapod configuration is used (a). While in this default configuration, the robot reacts to the sensors and locations. When the *Prey* sensor is set to true, the robot transforms to the *Cross* configuration and folds itself in place (b). Upon entering the *Tunnel* region, the robot transforms to the narrow and turning capable *Snake* configuration.

B. Scenario with Physical Robot

The task specification in structured English is shown below, for the workspace in Fig. 5. The traits of interest here are *hardware* (indicating configuration-gait pairs must be feasible with current hardware), *stationary, low* and *narrow*. Fig. 7 depicts snapshots of this example, and the accompanying video contains the entire specification being executed.

```
-Env starts with false
-Robot starts in Plains
-Always do T_hardware
-If you are not sensing Landslide and you are not sensing
Burn then visit Plateau
-If you are not sensing Landslide and you are not sensing
Burn then visit Plains
-If you were in Trail then do not Springs
-If you are not sensing Landslide and you are sensing
Burn then visit Springs
-Do T_stationary if and only if you are in Springs and
you are sensing Burn
-Always not Lava1 and not Lava2 and not Lava3
-If you were in between Lava1 and Lava2 then do not
sense Landslide
-T_low is set on between Lava1 and Lava2 and reset on
Plains or Plateau or Springs
-T_narrow is set on between Lava1 and Lava2 and reset
on Plains or Plateau or Springs
-If you are sensing Landslide then stay there
```

Because we cannot reconfigure instantaneously between configurations in hardware, especially if the number of modules changes drastically as in Section VI, we have used a single configuration with multiple gait sets. The *Tee* configuration (see Fig. 1) is similar to *Snake*, except it has two additional modules at each side of its "tail" module. The 'Snake.crawl' configuration-gait pair does not use the side modules and turns with the front "head" module. 'Tee.crawl', on the other hand, uses the side modules to turn but not the "head" module. When the robot is in *Snake* configuration it ignores the additional side modules.

The robot turns on the traits 'T_low' and 'T_narrow' in the *Trail* and *Volcano* regions (between Lava1 and Lava2), which the library maps to the *Snake* configuration-gait pair. The robot moves back and forth between *Plains* and *Plateau* unless it senses the *Burn* sensor. In this case the robot visits the *Springs* region and stays there until *Burn* is turned off. Once the robot is in the *Springs* region and if it is still sensing *Burn*, 'T_stationary' is activated which causes the robot to "swim" in the *TeeStationary.swim* configuration-gait pair – this is the same configuration as *Tee* but its gait is different. Also, unless the robot is in *Snake* configuration, it stays still while sensing *Landslide*.

VII. Conclusions and Future Work

In this paper we have demonstrated how tasks containing *traits* can be automatically transformed into provably correct control for a modular robot. We discussed the creation of a configuration-gait-trait library and tools that facilitate the process of creating entries for the library. The approach has been demonstrated both in simulation and with the CKBot hardware as seen in the paper and the accompanying video.

Our future work is focused on the following directions: (a) Enhancing the simulator by allowing the generation of gradients and uneven terrain inside regions to test the functionality of certain modular robot configurations. (b) Enriching the library (and therefore the task space) and automating the process of populating the library with methods similar to [24] and [25]. (c) Adding reconfiguration controllers to replace instantaneous robot transformations and (d) Experimenting with additional modular robot platforms.

Acknowledgments

We thank Jimmy Sastra and Dr. Mark Yim for providing us with the CKBot hardware, software and support.

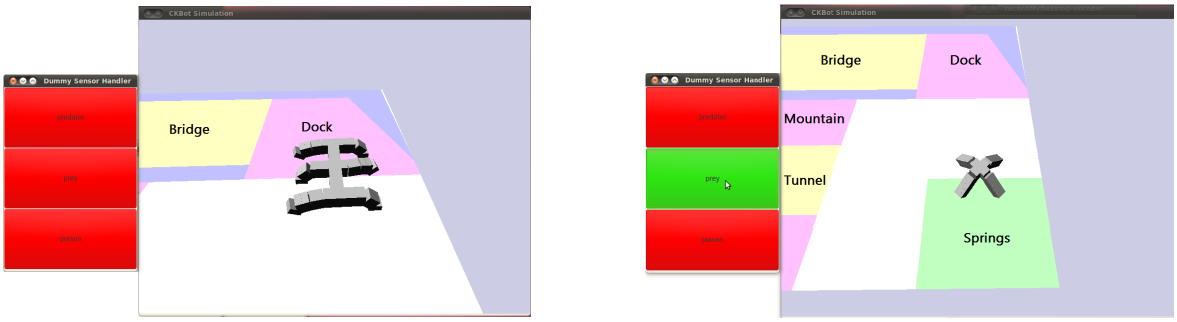

Fig. 6. Screenshots of the simulation scenario. [Left] The default Hexapod configuration moving in the environment. [Right] Cross Configuration activated when the Prey sensor (highlighted in green) becomes true.

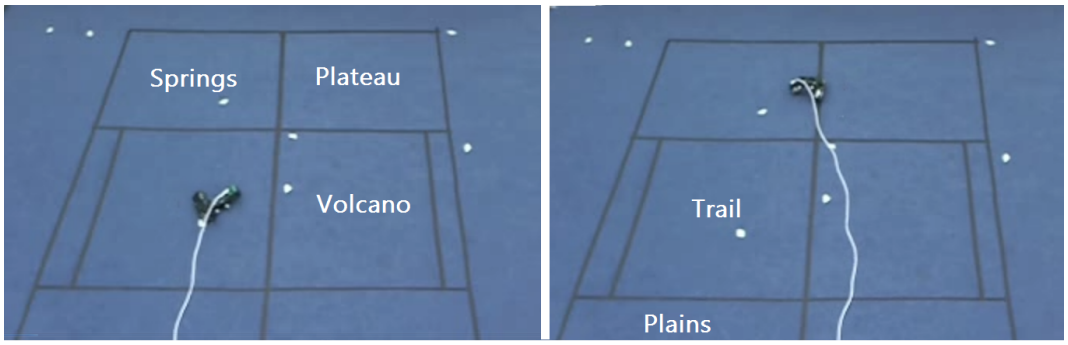

Fig. 7. Screenshots of the hardware scenario. [Left] Snake.crawl configuration-gait pair in between the Lava1 and Lava2 regions. [Right] TeeStationary configuration in the Springs region.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

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

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

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

[24] 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.

[25] 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.