

An End-to-End System for Accomplishing Tasks with Modular Robots

Gangyuan Jing Tarik Tosun Mark Yim Hadas Kress-Gazit
Cornell University Univ. of Pennsylvania Univ. of Pennsylvania Cornell University
gj56@cornell.edu tarik@grasp.upenn.edu yim@grasp.upenn.edu hadaskg@cornell.edu

Abstract—The advantage of modular robot systems lies in their flexibility, but this advantage can only be realized if there exists some reliable, effective way of generating configurations (shapes) and behaviors (controlling programs) appropriate for a given task. In this paper, we present an end-to-end system for addressing tasks with modular robots, and demonstrate that it is capable of accomplishing challenging multi-part tasks in hardware experiments. The system consists of four tightly integrated components: (1) A high-level mission planner, (2) A large design library spanning a wide set of functionality, (3) A design and simulation tool for populating the library with new configurations and behaviors, and (4) modular robot hardware.

The broader goal of this project is enabling users to address real-world tasks using modular robots. We believe this work represents an important step toward this larger goal.

I. INTRODUCTION

Modular reconfigurable robots have the ability to transform into different shapes to address a wide variety of tasks. They promise to be versatile, robust, and low cost [25]. Dozens of groups have built different kinds of reconfigurable robots [16, 11], and introduced approaches for programming them [19, 21, 26].

The advantage of modular robot systems over traditional systems is their flexibility: by changing configuration, they can in theory address a wider variety of tasks than a traditional robot of similar complexity. However, this advantage can only be realized if there exists some reliable, effective way of generating configurations (shapes) and behaviors (controlling programs) appropriate for a given task. At the time of writing, solving this problem remains a significant barrier to the application of modular robots to solve complex real-world tasks.

Recently, there has been increasing interest in automatic controller synthesis with correctness guarantees from high-level task specification in the form of formal languages [2, 3, 8, 10, 18, 24]. Most of its work has focused on traditional robot systems with a fixed configuration and a finite set of action capabilities, with the exception of [4] which presents a framework to synthesize controllers for a modular robot system from high-level task specifications.

Our goal in this work was to provide a system that enables users to solve complex tasks using modular robot hardware. Our approach is library-driven: rather than attempting to generate new designs from scratch, users specify task requirements and a design tool retrieves designs satisfying the

requirements from a library of existing useful designs. Our primary contribution is an end-to-end system that integrates low-level design generation, library management, high-level mission planning, and modular robot hardware. We present hardware experiments to demonstrate that it is capable of addressing challenging multi-part tasks, discuss the strengths and weaknesses of the system, and provide a roadmap forward to apply a similar system in a real-world setting.

A. System Overview

Before delving into the details, we provide a brief overview of the entire system. At the highest level, tasks are specified in a high-level mission planner using Linear Temporal Logic (LTL)[7, 10]. The user does not specify what configurations and behaviors should be used to complete the task, but rather describes the functionality required in terms of the task environment and desired behavior properties. For example: “if the robot is moving in a tunnel, maintain a maximum height of 3 units”.

To develop a solution to the task, the high-level mission planner fulfills each of the specified functionalities by automatically selecting robot configurations and behaviors from the design library. In a sense, the high-level planner treats the entire modular robot system as a single robot with a set of capabilities defined by the library.

Once the high-level mission planner has selected configurations and behaviors fulfilling all requested functionalities, the mission specification is compiled into a correct-by-construction finite state automaton that interfaces directly with the hardware, directing the modules to complete their task.

For this approach to be effective, the library must span a wide range of useful functionality. We created a physics-based simulator and design tool that allows users to easily build, program, and test modular robot designs. Any configuration or behavior created in the simulator can be directly ported to the hardware modular robot system, SMORES-EP.

All designs are saved to a web server, and are available to anyone who has the tool. We released the tool and held hackathons with volunteer engineering students, ultimately producing a library with 52 configurations and 97 behaviors which span a broad range of properties and environments.

B. Contributions and Paper Outline

Our primary research contribution is a complete system for accomplishing complex tasks with modular robots, with

four tightly integrated components: (1) A high-level mission planner, (2) A large design library spanning a wide set of functionality, (3) A design and simulation tool for populating the library with new configurations and behaviors, and (4) modular robot hardware. Additionally, we introduce a theoretical contribution in high-level control of modular robots by checking the feasibility of implementing desired robot behaviors prior to the controller synthesis.

The remainder of this paper is structured as follows. In Section II, we present related work. In Sections III, we present the theoretical framework underlying our high-level mission planner and design library. In Section IV, we present the four system components mentioned previously. In Sections V, we present three experimental case studies (two in hardware, one in simulation). In Section VI and evaluate the effectiveness of our system and discuss future work. Finally, in Section VII, we conclude.

II. RELATED WORK

Mehta *et al.* [13, 14] present tools allowing novice users to create and print robots from high-level specification. The focus of their work is on synthesis of electromechanical systems and low-level controllers, primarily addressing low-level functionality (such as the ability to walk or grip) rather than high-level, multi-part tasks situated in an environment. Schulz *et al.* [20] present tools for designing foldable robots and a library of robots created by volunteers. In [23], Tosun *et al.* introduce a physics-based simulator, design creation tool, and a small hierarchically organized library for the SMORES robot.

Castro *et al.* [4] introduce high-level control for the CKBot modular robot, and lay the theoretical foundation for our high-level mission planner. Our work differentiates itself from [4] in several ways. We expand the notion of behavior properties to include both behavior and environmental properties, increasing the expressiveness and granularity of task specification. In terms of theory, we also introduce a performance improvement by grounding actions in concrete configurations and behaviors prior to automata synthesis. More significant than the theoretical differences is the larger scope and capability of our system. Where Castro *et al.* present a small library (8 designs, 19 behaviors, 7 properties) and a basic design/simulation tool, we present a design large library (52 designs, 97 behaviors, 19 properties) and a sophisticated cross-platform design tool, intended to allow many contributors to build a large, useful library.

In addition, where Castro *et al.* perform basic hardware experiments (using a single CKbot configuration to visit regions on the ground), we have performed more significant hardware experiments to verify our system. In this paper, we present two hardware case studies using SMORES-EP modules to complete multi-stage tasks consisted of traversing challenging environment, manipulating objects, and disconnecting and reconnecting modules. We also link our high-level mission planner to our physics-based simulator, allowing automatic

controller synthesis from a library of robot behaviors. The synthesized controllers are demonstrated in our simulator.

Several simulators are available for designing robot controllers, including simulators specifically for modular robots [5], and Gazebo [9] which is a popular physical simulator. For this work, we opted to use Unity3D [1] because it is cross-platform and offers more tools to create user interfaces, which proved very important to making it easy to create new designs and behaviors. Unity also offers good physics-based simulation.

III. BACKGROUND

In this section we define basic terms related to modular robot systems.

A. Definitions

Definition 1 (Module): A module is the basic unit of a modular robot system. It is a small robot that can respond to commands, move, and connect to other modules. We define a module as $m = (J, A)$. $J = \{J_1, \dots, J_d\}$ is set of joints of the module with d degrees of freedom. $A = \{A_1, \dots, A_k\}$ is the set of *attachment points* where the module can connect to other similar modules. We denote the attachment point A_i of module m as $m.A_i$.

Definition 2 (Command): A command to a joint J_i is defined as $u_{J_i} = (\alpha, V, t)$, where $\alpha \in \{\text{Position, Velocity}\}$ is the type of the command. $V \in \mathbb{R}$ is the value of the command and $t \in \mathbb{R}$ is the time period of the command. For example, the joint can be commanded to hold joint position $V = \frac{\pi}{2} \text{ rad}$, or to maintain the joint velocity $V = \pi \frac{\text{rad}}{\text{sec}}$ for time duration t . A joint stops moving once it reaches the desired joint position or the time period has passed. We assume there are low-level controllers (*e.g.* PID controllers) that can drive the corresponding joint to satisfy the command u_{J_i} .

Definition 3 (Configuration): A configuration is a set of connected modules. In this work, we treat a configuration as a single robot. We define a configuration as $\mathcal{C} = (M, E)$, where $M = \{m_1, \dots, m_q\}$ is the set of modules in the configuration. E includes pairs of attachment points. $(m_{i_1}.A_{i_2}, m_{j_1}.A_{j_2}) \in E$, where $m_{i_1}, m_{j_1} \in M$, and $m_{i_1} \neq m_{j_1}$. In this work, we require that each configuration has only one independent part (it cannot have two disconnected sets of modules that move independently).

Definition 4 (Behavior): A behavior $B_C = b_1, \dots, b_n$ is a sequence of behavior states and is associated with a configuration C . Each behavior state is defined as $b_i = (U, T)$. U is the set of joint commands for all joints in the configuration: for the given configuration C , we have $u_{J_i} \in U$, for all $J_i \in J$ and for all $m \in M$. The time period of the behavior T is defined as the largest time period of all joint commands in U .

IV. SYSTEM

A. Modular Robot Hardware - SMORES-EP Robot

Our system is built around the SMORES-EP robot, but it could easily be adapted to work with other hardware platforms. In this section, we provide a brief introduction to the technical

capabilities of SMORES-EP. A more detailed discussion is out of the scope of this paper, and is being published in parallel. The SMORES-EP system is kinematically equivalent to the earlier version SMORES [6] system but includes a new electro-permanent latching system. Readers are referred to [6] for more detail.

Each module has four DoF - three continuously rotating faces (left, right, and pan) and one central hinge (tilt) with a 180° range of motion (Fig. 1). The DoF marked left, right, and tilt have rotational axes that are parallel and coincident. A single module can use its left and right wheels to drive around as a two-wheel differential drive robot. All four faces of the SMORES-EP module have electro-permanent (EP) magnets that serve as low-power, hermaphroditic connector for self-reconfiguration. Any face of one module can connect to any face of another.

Some of the motions a SMORES-EP cluster can perform are limited by the strength of the magnetic connectors, which can support the weight of at most three modules cantilevered horizontally against gravity. This limitation is alleviated in some cases by using rigid connector plates, which are screwed into the faces of two modules to create a strong permanent connection between them. Using connector plates, up to four modules can be cantilevered before exceeding the torque limits of the motors. However, because the connector plates must be manually screwed into place, modules with connector plates cannot self-reconfigure.

Each module has an onboard battery, microcontroller, and 802.11b wireless module to send and receive UDP packets. In this work, clusters of SMORES modules were controlled by a central computer running a Python program that sends wireless commands to control the four DoF and magnets of each module. Battery life is about one hour (depending on motor, magnet, and radio usage), and commands to a single module can be received at a rate of about 20hz. Wireless networking was provided by a standard off-the-shelf router, with a range of about 100 feet.

The cluster is localized using AprilTags [15] mounted on one or more modules. In our experiments, AprilTags were also mounted on objects of interest in the environment. The AprilTag tracking software, high-level planner, and SMORES-EP cluster control software were all run simultaneously on a Dell laptop (2.4GHz, 4Gb of RAM) with an overall control loop time of about 4Hz (limited by the AprilTag detection software).

B. Design and Simulation Tool - VSPARC

To populate the robot design library, we used the Unity3D Engine [1] to design and implement a software tool called VSPARC, which stands for **V**erification, **S**imulation, **P**rogramming And **R**obot **C**onstruction. It allows users to design configurations and behaviors for the SMORES-EP modular robot system, and simulate behaviors with a physics engine. Moreover, users can save and share their designs online, which enables us to use VSPARC for crowdsourcing

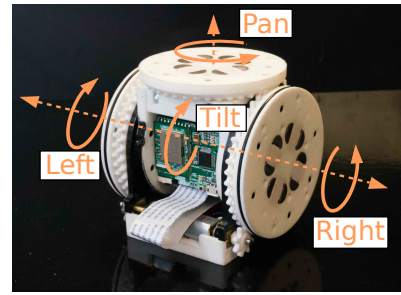


Fig. 1: SMORES-EP module

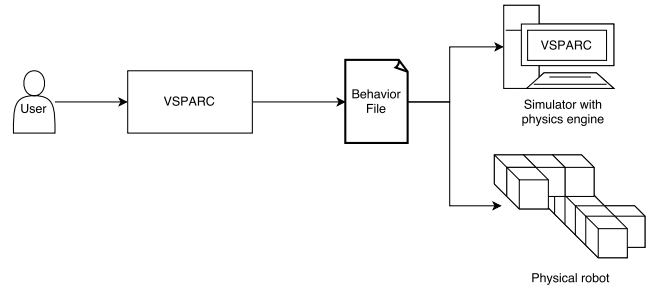


Fig. 2: The same behavior file can be used by both the simulator and the physical robot.

the robot design library. VSPARC’s main features are listed below:

- Design configurations with unlimited number of modules and visualize the design in a 3D environment.
- Position and velocity control of all module joints.
- Design behaviors for any configuration by creating a sequence of joint commands.
- Simulate the performance of any behavior in a physics engine.
- Create and share designs online. Test and improve other users’ designs.

Behaviors designed in VSPARC can be exported as XML files, which can be easily run to control the physical robot.

C. Design Library

Definition 5 (Property): A property is a high-level descriptor of the robot behavior or the environment. A property is defined as $p = (p_n, \Omega)$, where p_n is in an English description as the name of the property. Ω is the set of values of the property. For example, a robot property $p = (\text{Action}, \{\text{Move}, \text{Push}\})$ means the robot behavior can perform both *Move* and *Push* actions. We can also use properties to describe the environment in which the robot operates. For example, a property $p = (\text{BoxMass}, [2, 5])$ means the mass of a box in the environment could be any value between 2 and 5 units. In this case, the property is a quantitative description of the environment. In Table I, we list some examples of property names for common robot tasks.

TABLE I: Examples of property names

Properties for Robot Behavior	Properties for Environment
Speed	Box_Mass
Width	Stair_Height
Height	Ground_Roughness
Action	Tunnel_Height

We say a property $p_1 = (p_{n_1}, \Omega_1)$ satisfies a property $p_2 = (p_{n_2}, \Omega_2)$ if and only if $p_{n_1} = p_{n_2}$ and $\Omega_1 \subseteq \Omega_2$.

Definition 6 (Robot Design Library): A robot design library for a specific modular robot system is a collection of configurations and behaviors labeled with environment and robot behavior properties. The library $\mathcal{L} = \{l_1, \dots\}$ consists of a set of library entries. A library entry is defined as $l = (C, B_C, P_e, P_r)$, where C is the configuration and B_C is a behavior associated with C . P_e and P_r are sets of properties that describe the environment and robot behavior respectively. For example, a library entry $l = (C_{\text{snake}}, B_{\text{climb}}, P_e, P_s)$, where $P_e = \{(\text{Stair_Height}, [2, 3])\}$ and $P_s = \{(\text{Action}, [\text{Climb}]), (\text{Speed}, [1])\}$, represents a `snake` shape configuration with a `climb` behavior that can climb a stair with the height of two or three units, with the speed of 1 unit. Moreover, we say a library entry l satisfies a property p if there exist a property $p' \in P_e \cup P_r$ such that p' satisfies p .

To generate the configurations and behaviors in the library, we made our design tool available online at www.vsparc.org. We distributed the tool to undergraduate and graduate student volunteers, and hosted three hackathons in which participants created designs and behaviors for several hours. The library includes 52 designs and 97 behaviors contributed by 20 volunteers. Since the full library is too large to list in this paper, we provide a representative sampling of configurations, behaviors, and properties in Fig. 3. The unit for length is the side length of a single SMORES-EP module. The unit for mass is the mass of a single SMORES-EP module.

D. Reactive Controller Synthesis and Execution with the Library

To fully exploit a design library with a wide collection of robot configurations and behaviors with properties over the environment and the robot, we want to be able to specify robot tasks from a high-level perspective and automatically generate robot controllers that utilizes library entries to satisfy desired tasks. In this section, we introduce a system to incorporate library entry searching into a reactive controller synthesis and execution framework [7, 10] using LTL as the formal language for task specifications.

To automatically generate a robot controller from high-level task specification, the robot workspace, robot action capabilities and environment events are first abstracted into sets of atomic propositions. Reactive robot tasks, where the robot needs to respond to different environment events, are expressed by LTL formulas over those propositions. The synthesis algorithm can then determine the existence of a controller satisfying given robot tasks. If such a controller exists, it can be automatically generated in the form of a finite

state automaton. Finally, the discrete automaton is executed to control the robot to achieve desired tasks. This framework is implemented in an open source software package: LTLMoP (Linear Temporal Logic Mission Planning) Toolkit [7].

In [7], robot action capabilities are first abstracted into a set of *robot* propositions. Then the synthesis algorithm automatically generates a finite state automaton, if possible, from a given task specification in LTL. Lastly, a user will map each *robot* proposition to a low-level robot controller so that the robot will execute the desired action whenever the corresponding proposition becomes `True`.

In this work, instead of manually abstracting robot action capabilities, a set of library entries in the existing library is automatically mapped with an atomic proposition (Section IV-D1). Moreover, the matched entries are used to automatically create additional constraints between the set of *robot* propositions in the form of LTL formulas (Section IV-D2). The additional LTL formulas are appended to the original task specification prior to synthesis. The synthesized finite state automaton, if it exists, can be used to control both the simulated or the physical robot using matched library entries. This framework is illustrated in Fig. 4.

1) *Match library entries with propositions:* We denote the set of robot atomic action propositions as Act representing the action capabilities of the robot. For example, if the proposition `push` is `True` it means the robot is activating the push action. For each action propositions $y \in Act$, we allow the user to assigns a set of environment and robot properties $P_y = \{p_1, \dots, p_n\}$ as defined in Definition. 5. Then we can search through the existing robot design library to find a set of library entries $L_y = \{l_1, \dots, l_k\}$ that satisfies all property in the set P_y .

2) *Generate and append additional LTL formulas to the original task specification:* Once we match each robot action proposition with a set of library entries, we have a relation $\lambda : Act \rightarrow 2^{\mathcal{L}}$ that maps each propositions $y \in Act$ to a set of library entries L_y that satisfies the user specified set of properties P_y for y . We say a library entry l can *implement* a proposition y if $l \in \lambda(y)$. For any $y \in Act$, if $\lambda(y) = \emptyset$, we need to make sure proposition y is never `True`, because no library entry can implement y . For any $y, y' \in Act$, if $\lambda(y) \cap \lambda(y') = \emptyset$, we need to make sure proposition y and y' can never be `True` at the same time, because there does not exist a library entry that can implement both y and y' . To encode the mutual exclusion between robot propositions into the task specification, we specify them in the form of LTL formulas and take them into account during the synthesis.

3) *High-Level Mission Planner - LTLMoP:* LTLMoP [7] is a Python-based toolbox that allows users to specify robot tasks in Structured English or LTL formulas and automatically synthesizes a correct-by-construction controller, if it exists, that satisfies given tasks. In addition, LTLMoP can also execute the synthesized controller with a wide range of supported robot systems. We incorporate proposition mapping and LTL generation to LTLMoP to allow controller synthesis from high-level task specifications with a modular robot design







Behavior Name	Single module	Rolling Loop	DoubleDriver	Stair Climber	Swerve Driver	backhoe
Number of modules	1	8	7	4	9	9
						
Locomotion						
Max robot height	1	2.5	1.5	2	2	4
Max robot width	1	1	3	1	4	4
Max robot length	1	5	3	3.5	3	7
Terrain - Smooth	✓	✓	✓	✓	✓	
Terrain - Rough		✓		✓		
Terrain - Sloped		✓	✓	✓		
Driving - Straight	✓	✓	✓	✓	✓	
Driving - Differential drive	✓		✓	✓	✓	
Driving - Holonomic					✓	
Ledge ascent - Max height		0.25		0.75		
Ledge ascent - # modules lifted		all		all		
Ledge descent - Max height		1		1.5		
Ledge descent - # modules lowered		all		all		
Manipulation						
Attachment - Push	✓	✓	✓	✓	✓	✓
Attachment - Magnetic	✓		✓	✓		✓
Attachment - Carry					✓	
Workspace size	X : [- inf, inf] Y : [- inf, inf] Z : [0, 1]	X : [- inf, inf] Y : [0, 1] Z : [0, 2.5]	X : [- inf, inf] Y : [- inf, inf] Z : [0, 1.5]	X : [- inf, inf] Y : [- inf, inf] Z : [0, 2]	X : [- inf, inf] Y : [- inf, inf] Z : [0, 2]	X : [-3, 3] Y : [-3, 3] Z : [0, 4]
Payload mass	1	2	4	2	3	1

Fig. 3: Matrix of designs and properties. (Unit Length = the side length of a single SMORES-EP module. Unit Mass = the mass of a single SMORES-EP module)

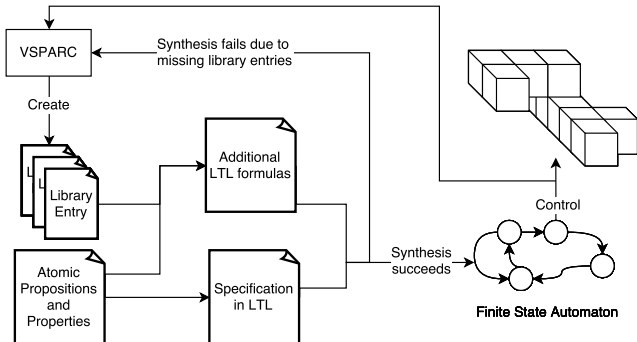


Fig. 4: Controller synthesis and execution as described in this work.

library. Fig. 4 shows the framework of the modified LTLMoP. A user can specify a set of properties for each robot action proposition with a file in Extensible Markup Language (xml) format. LTLMoP will parse this file together with the robot design library and generate a set of additional LTL formulas that encodes the mutual exclusive relations between all robot action propositions. These LTL formulas are combined with the original robot task specification from the user prior to the controller synthesis. If the synthesis is successful, a discrete robot controller will be generated. When executing the synthesized controller, if there are multiple library entries matching a proposition, we choose one randomly. If the synthesis

failed with the additional LTL formulas, possibly due to lack of library entries that implements some action propositions, LTLMoP will notify the user who can then design those missing library entries with VSPARC.

V. EXPERIMENTAL RESULTS

Here, we present three case studies to illustrate the capabilities of our system.

A. Simulated Demos

In these simulated scenarios, we demonstrate the capabilities of the system by showing different robot controllers to address the same user-defined task specification in different environment setups.

1) *Scenario 1*: The environment for Scenario 1 is shown in Fig. 5. It consists of a button, a lightweight block, a gap in the ground, and a ramp, all in a straight line. The objective is for the robot to move from its starting point to the goal area at the top of the ramp. When the button is pushed, it causes the block (which begins floating in the air) to fall to the ground, where it can be pushed into the gap, forming a bridge between the flat region and ramp. The action definitions for this task are provided below:

```

pushButton:  type = Manipulation_Push
              height = 1.5
pushBox:     type = Manipulation_Push
              payload = 2
              distance_x = 3
climb:       type = Locomotion
              drive = Straight
              terrain = Sloped

```

The high-level mission planner searches the library for one or more entries that fulfill all requested properties. In this case, all the properties are fulfilled by the `rollingLoop` configuration, so no reconfiguration is needed to complete the task. In the top row of Figure 9 and in the accompanying video, we see the rolling loop complete the task.

2) *Scenario 2*: The environment for Scenario 2 is also shown in Fig. 5. It is similar to the environment from Scenario 1, but with several small changes that make the task more difficult. The button is now to the side of the map, and floats at a height of 4 module-lengths above the ground. The box is twice as heavy, weighing 4 module-weights rather than 2. The ramp has been replaced with stairs with the step height of 0.75. The action definitions are provided below:

```

pushButton:  type = Manipulation_Push
              height = 4
pushBox:     type = Manipulation_Push
              payload = 4
              distance_x = 3
climb:       type = Locomotion
              drive = Straight
              ledge height = 0.75

```

The `rollingLoop` can no longer complete this task - it can't reach the button, it's not strong enough to push the box, and it can't ascend steps higher than 0.25. When the specification is compiled, our system selects three different configurations from the library to complete the task. The `backhoe` is used to push the button, because it is the only configuration with a large enough vertical workspace. It then reconfigures into the `doubleDriver` to drive over and push the box into the hole, because it is able to drive, turn, and push objects as heavy as 5 module-weights. Then, it reconfigures into the `stairClimber` to climb the stairs. In the bottom row of Figure 9 and the accompanying video, we see how the task is completed. For the purposes of this paper, we assume that reconfiguration between any two configurations is possible as long as the final configuration does not have more modules than the initial configuration.

B. Hardware Demos

In these scenarios, a cluster of SMORES modules is directed to clean the top of a table. The map is shown in Fig. 6. There are two components to this task: first, the robot must move a waste bin near the table, and then, the robot must climb up to the top of the table, explore the surface, and react appropriately to the objects it encounters. These two scenarios showcase the translation of behaviors from the simulator to hardware, and the ability to use LTLMoP to create and execute mission plans with the hardware and AprilTag sensing in the loop.

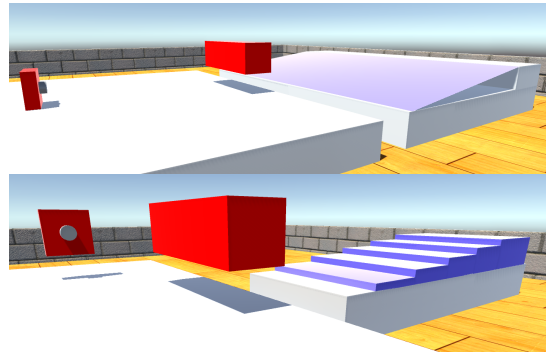


Fig. 5: Environments for Scenarios 1 (top) and 2 (bottom) in the simulator.

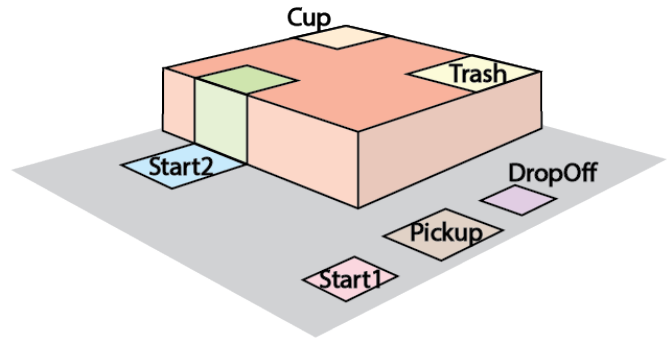


Fig. 6: Map of the hardware demo

In order to limit the number of hardware configurations we would need to use for these demos, we opted to hand-select the behaviors to use as actions in our mission specification. If we had wanted, we could have instead specified the actions in terms of properties and environments, and allowed LTLMoP to ground them in configurations and behaviors from the library.

1) *Moving the Waste Bin*: In the first scenario, the SMORES cluster starts in region `Start1` and needs to move a waste bin from region `Pickup` to region `DropOff` (a distance of 10 module lengths) to be near the table, and then travel to the table edge (region `Start2`). The task is made more difficult by the fact that the waste bin is supported by four legs, so it cannot be pushed by designs with a height of two modules or less. The workspace requirement (10 modules lengths) rules out all stationary manipulators in the library, and height requirement rules out most car-like designs. Fortunately, the `swerveLifter` design, which can carry objects by driving under them and lifting up, is perfect for the job.

The high-level task is specified as follows:

- `carry` is set on `pickup` and reset on `false`
- `dropped` is set on `drop` and reset on `false`
- do `pickup` if and only if you were sensing `wasteBin` and you are not activating `carry`
- do `goToTable` if and only if you are activating `dropped`
- do `drop` if and only if you were activating `carry` and you are not activating `dropped`

As seen in the accompanying video and Figure 8, the resulting state machine directs the robot to complete the task. The robot waits until it senses the waste bin (marked with an AprilTag). Then it lowers itself, drives under the waste bin, carries it next to the table, and executes a swerve-driving (small-time holonomic) behavior to travel to the table.

Overall, this scenario is a success, with the high-level planner successfully sequencing library behaviors in response to the presence of the waste bin. However, we faced several challenges in completing this task with our system. First, magnetic connector strength proved problematic, and we had to use a passive module in the middle of the structure in order to perform the sit-down and stand-up behaviors without breaking off one or more legs. Second, because the `swerveLifter` steers by aligning four caster wheels in the same direction its performance is sensitive to encoder calibration errors in the DoF controlling wheel direction, and in the video we see the `swerveLifter` drift from its intended straight-line path. The video shown is one of nine takes, with most failures caused by sensor calibration issues.

2) *Table Exploration:* With the waste bin in place, the cluster can clean the top of the table. It should explore the tabletop and react to what it finds: if it senses a piece of trash, it should push it off the table, and if it senses a coffee mug, it should back up and spin in place (alerting the mug’s owner that it should be removed). After exploring both locations on the table, it should return to the ground.

We selected to include behaviors from the `snake7` and `module1` configurations in the spec for this mission. The `snake7.climbup` behaviors allows the robot to lift its front two modules up to the top of the table. Once there, the front module can use `undock` to disconnect from the rest of the body, allowing it to function independently as a `module1` configuration and use the `differentialDrive`, `spin`, and `push` behaviors. When the robot is ready to come back down, it can use `dock` to re-connect to the body, and the `snake7.climbdown` behavior to return to the floor.

The high-level mission spec is shown below:

- if you are sensing **mug** then do **spin**
- if you are sensing **trash** then do **push**
- **loc1visited** is set on **loc1** and reset on false
- **loc2visited** is set on **loc2** and reset on false
- do **docking** if and only if you were in **dock** and you are activating (**loc1visited** and **loc2visited**)
- do **undock** if and only if you were in **dock** and you are not activating (**loc1visited** or **loc2visited**)
- do **climbdown** if and only if you were in **dock** and you activated (**loc1visited** and **loc2visited**)
- do **climbup** if and only if you were in **ground** and you are not activating (**loc1visited** or **loc2visited**)
- infinitely often do **docking**

The resulting state machine directs the robot to successfully clean the table, as shown in the accompanying video and Fig. 7,8. An overhead camera system tracks AprilTags attached to the first module of the snake, allowing LTLMoP to servo the left and right wheels of the module in differential drive

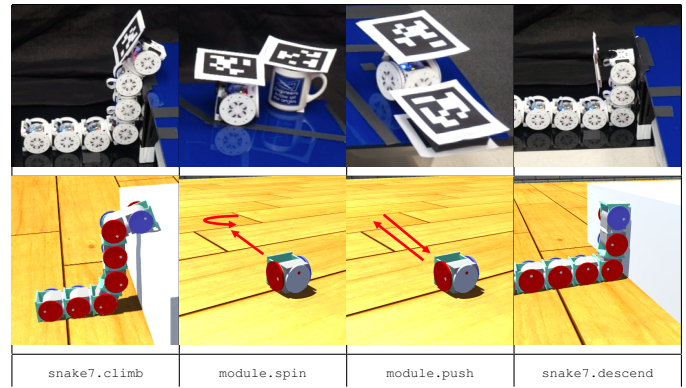


Fig. 7: Cleaning the Table

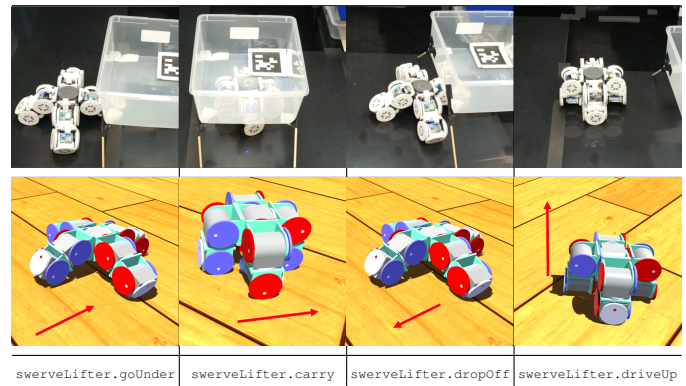


Fig. 8: Moving the Waste Bin

and sense proximity to the coffee mug and trash (also marked with AprilTags).

As with the waste bin moving scenario, we faced several challenges when solving this task using our system. The `climbUp` behavior is open loop, and not robust to initial robot position - in several trials, we started the snake too close to the ledge, causing it to jam its head into the corner of the ledge and break. Autonomous re-assembly of the snake head and body was the least reliable part of the experiment, succeeding about 25% of the time. We believe this could be significantly improved in the future by implementing a docking procedure that includes error checking and multiple re-tries, similar to the one used in [17].

Magnetic connector strength once again presented a challenge in this scenario. In the video, we see the snake break in the middle during its descent, and then re-connect to complete the descent successfully. This was sporadic, likely due to modules on the top of the table catching on the edge as they are pulled down. This kind of problem is difficult to predict in the simulator.

The video we present is one of 13 takes, with most failures caused by poor initial positioning of the snake and failure to re-connect the head. Once the proper initial position was found, the entire mission ran fairly repeatably.

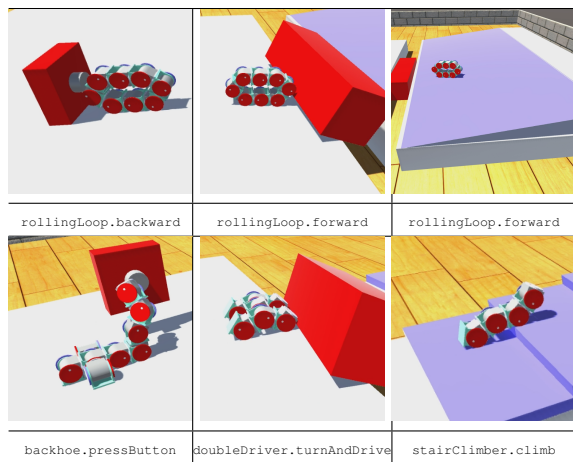


Fig. 9: Simulated Demo

VI. DISCUSSION AND FUTURE WORK

A. Simulator-to-hardware translation

Prototyping designs and behaviors in the simulator resulted in significant time savings over prototyping in hardware. The primary difficulties in translating behaviors from the simulator to the hardware lay in hardware issues the simulator did not reflect, *i.e.* connection strength and encoder calibration errors. Future work includes modeling connector strength and encoder errors in the simulator, to identify problems without doing a hardware test.

A major avenue of future work is the incorporation of sensing. We are working on a sensor module with an RGB-D camera for the SMORES robot. This sensor information will allow our system to autonomously complete tasks in new environments.

B. Composing Library Elements to Complete Missions

Environment and behavior properties provide an expressive way for the user to specify the requirements of a task. However, the fact that a behavior is labeled with a specific property does not guarantee it will perform as intended in all circumstances. Adapting open-loop behaviors to environments different from the one in which they were designed can cause them to fail, as evidenced by the problems in establishing proper initial robot position for the `climbUp` behavior in the table cleaning scenario. Developing methods for automatically analyzing tasks and environments is a field of active research [22]. Determining optimal sets of environment factors and integrating methods for automatic task analysis and would be an interesting avenue for future work. Introducing closed-loop behaviors that use sensor feedback is also future work.

It's worth noting that some behaviors are much more tolerant to varying environments than others. In our hardware experiments with the `stairClimber` configuration, we found that a single open-loop gait was able to climb steps of several varying sizes with no problems. Establishing confidence bounds on behavior success as a function of

environment parameters and including this information in the library is future work.

C. Low-level behavior creation

Developing sophisticated designs and behaviors in the simulator requires skill and experience. In our experience with the hackathons, undergraduate engineers became significantly faster and more adept at creating designs and behaviors through hours of practice. Newcomers spent on average about one hour creating a useful behavior, while experienced users would spend about twenty minutes.

Most users required only a few minutes to build a new configuration and conceive the fundamental motions it should perform. Most of the design time was often spent coding joint trajectories to achieve the desired motion without violating constraints, such as maintaining balance and avoiding connector strength overload. In the future, we hope to incorporate motion planning tools and allow users to specify desired motions without explicitly coding joint angles. Evolutionary techniques will be explored to generate behaviors automatically.

We also plan to automatically generate behaviors for new configurations with substructures that embed the kinematics of existing configurations [12]. The same embedding detection algorithm could also be used to port behaviors from SMORES to another modular robot system, or vice-versa.

VII. CONCLUSION

In this paper, we present (1) a physics-based simulator that allows creating and managing a modular robot design library with multiple users; (2) a framework for labeling each entry in the library with descriptive properties; (3) a connection to a reactive controller synthesis tool for automatically generating robot controllers from the library; and (4) experiments with both simulated and physical modular robots that demonstrate the performance of generated controllers. At the low-level, the tedious, error-prone process of designing configurations and behaviors for a modular robot system is simplified by our growing modular robot design library. At the high-level, controller synthesis with formal language provides guarantees on the correctness of synthesized controllers. The seamless connection between the high-level and low-level aspects results in a useful end-to-end system for controlling modular robot systems to accomplish user specified tasks.

The system still has limitations. Better modeling of the modular robot system, such as connector strength and joint encoder error, is needed to bridge the gap between the simulator and the physical hardware. A more rigorous method for expanding and consolidating properties in the library is necessary to guarantee the library covers a wide range of capabilities.

Overall, this work furthers the broader goal of this project and makes progress towards building a correct, robust, and accessible system that allows users to accomplish real-world tasks using modular robots.

ACKNOWLEDGMENTS

This work was funded by NSF grant numbers CNS-1329620 and CNS-1329692.

REFERENCES

- [1] Unity3d. <http://unity3d.com/>. Accessed: 2015-04-35.
- [2] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G.J. Pappas. Symbolic planning and control of robot motion [grand challenges of robotics]. *Robotics Automation Magazine, IEEE*, 14(1):61–70, March 2007.
- [3] Amit Bhatia, Lydia E Kavraki, and Moshe Y Vardi. Sampling-based motion planning with temporal goals. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2689–2696. IEEE, 2010.
- [4] Sebastian Castro, Sarah Koehler, and Hadas Kress-Gazit. High-level control of modular robots. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3120–3125. IEEE, 2011.
- [5] David Christensen, David Brandt, Kasper Stoy, and Ulrik Pagh Schultz. A unified simulator for self-reconfigurable robots. In *IROS*, 2008.
- [6] Jay Davey, Ngai Kwok, and Mark Yim. Emulating self-reconfigurable robots-design of the smores system. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4464–4469. IEEE, 2012.
- [7] C. Finucane, Gangyuan Jing, and H. Kress-Gazit. Ltl-mop: Experimenting with language, temporal logic and robot control. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1988–1993, Oct 2010.
- [8] M. Kloetzer and C. Belta. A fully automated framework for control of linear systems from temporal logic specifications. *Automatic Control, IEEE Transactions on*, 53(1):287–297, Feb 2008.
- [9] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IROS 2004*, 2004.
- [10] Hadas Kress-Gazit, Gerogios E. Fainekos, and George J. Pappas. Temporal logic based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009.
- [11] Haruhisa Kurokawa et al. Distributed self-reconfiguration of m-tran iii modular robotic system. *IJRR*, 2008.
- [12] Yannis Mantzouratos*, Tarik Tosun*, Sanjeev Khanna, and Mark Yim. On embeddability of modular robot designs. In *IEEE International Conference on Robotics and Automation*, 2015.
- [13] AM Mehta, Nicola Bezzo, B An, P Gebhard, Vijay Kumar, Insup Lee, and Daniela Rus. A design environment for the rapid specification and fabrication of printable robots. In *14th International Symposium on Experimental Robotics (ISER14), Marrakech/Essaouira, Morocco, June*, pages 15–18, 2014.
- [14] Ankur M Mehta, Joseph DelPreto, Benjamin Shaya, and Daniela Rus. Cogeneration of mechanical, electrical, and software designs for printable robots from structural specifications. In *Intelligent Robots and Systems (IROS 2014)*, pages 2892–2897. IEEE, 2014.
- [15] Edwin Olson. Apriltag: A robust and flexible visual fiducial system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3400–3407. IEEE, 2011.
- [16] Esben Hallundbæk Østergaard, Kristian Kassow, Richard Beck, and Henrik Hautop Lund. Design of the atron lattice-based self-reconfigurable robot. *Autonomous Robots*, 21(2):165–183, 2006.
- [17] James Paulos, Nick Eckenstein, Tarik Tosun, Jungwon Seo, Jay Davey, Jonathan Greco, Vijay Kumar, and Mark Yim. Automated self-assembly of large maritime structures by a team of robotic boats.
- [18] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M. Murray, and Sanjit A. Seshia. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC '15*, pages 239–248, New York, NY, USA, 2015.
- [19] Behnam Salemi, Wei-Min Shen, and Peter Will. Hormone-controlled metamorphic robots. In *ICRA*, 2001.
- [20] Adriana Schulz, Cynthia Sung, Andrew Spielberg, Wei Zhao, Yu Cheng, Ankur Mehta, Eitan Grinspun, Daniela Rus, and Wojciech Matusik. Interactive robogami: data-driven design for 3d print and fold robots with ground locomotion. In *SIGGRAPH 2015: Studio*, page 1. ACM, 2015.
- [21] Kasper Stoy, Wei-Min Shen, and Peter M Will. Using role-based control to produce locomotion in chain-type self-reconfigurable robots. *Mechatronics, IEEE/ASME Trans. on*, 2002.
- [22] Jaeyong Sung, Seok Hyun Jin, Ian Lenz, and Ashutosh Saxena. Robobarista: Learning to manipulate novel objects via deep multimodal embedding. *arXiv preprint arXiv:1601.02705*, 2016.
- [23] Tarik Tosun, Gangyuan Jing, Hadas Kress-Gazit, and Mark Yim. Computer-aided compositional design and verification for modular robots. In *International Symposium on Robotics Research*, 2015.
- [24] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC '10*, pages 101–110, New York, NY, USA, 2010.
- [25] Mark Yim, David G Duff, and Kimon D Roufas. Polybot: a modular reconfigurable robot. In *ICRA*, 2000.
- [26] Ying Zhang et al. Phase automata: a programming model of locomotion gaits for scalable chain-type modular robots. In *IROS*, 2003.