

Computer-aided design of complex configurations and behaviors for modular robots

Author Names Omitted for Anonymous Review. Paper-ID [add your ID here]

Abstract—In this paper, we present a software framework for the compositional design of modular robot configurations and behaviors. Designs are constructed hierarchically by composing elements from a library, allowing users to easily create complex designs. Likewise, complex behaviors are constructed by composing controllers from a library in a nested series/parallel structure. The system is integrated with a full dynamic simulator, and provides tools to identify common problems with behaviors, specifically self-collision and loss of quasi-static stability.

I. INTRODUCTION

Modular reconfigurable robot systems have been studied extensively for several decades **TODO: citations**. These systems distinguish themselves in their ability to transform into different shapes to address a wide variety of tasks.

This additional flexibility places an additional burden on the user, because solving problems with modular robots involves not only designing software, but also the best physical form for the task at hand. We argue that if this complexity is not appropriately managed, it can make the system impractical. If the user is free to create any new design he/she pleases to solve a new task, but must program the design from scratch every time, creating new designs will be a huge amount of effort, and the advantage of flexible modular hardware will be defeated.

Software modularity is a well-established practice for developing large maintainable systems and avoiding duplication of effort **TODO: cite**. In robotics, software behaviors are inextricably linked to the hardware they control, resulting in additional challenges to modularity. Significant progress has been made in sharing robotics software between researchers and hardware platforms, most notably ROS which provides IPC and standard libraries for common robot tasks.

The challenges of software design are different in modular robotics than in typical robotics, because the hardware itself is modular. Porting software from one robot platform to a completely different robot platform takes a considerable amount of time, and needs to be facilitated by a large framework such as ROS, in which fundamental software libraries are almost totally decoupled from specific hardware. In modular robotics, the emphasis needs to be on speed of design, because the major advantage of a modular robot system is that new designs can be made for each task. We need to very quickly develop simple kinematic behaviors (*i.e.* take a step with a leg) with new morphologies that share some of the structure of old morphologies.

Our solution to this problem is to let the modularity of our hardware determine the modularity of our software. We

create new modular robot designs by combining existing sub-designs, for example combining four legs with a body to create a walking robot. We provide a GUI tool that allows users to do this easily. Designs have associated libraries of software behaviors, so that when new designs are created by composing existing sub-designs, new behaviors for that design can be quickly and easily created by composing the behaviors associate with its component sub-designs. We introduce new scripting language with a series-parallel execution structure that allows old behaviors to be easily and clearly combined into new behaviors.

Since combining old things in new ways can lead to unexpected problems, we also need to verify that our new designs and behaviors perform the way we expect them to. This is done in a dynamic simulation in Gazebo, and through verification tools that detect common problems.

II. CONTRIBUTION AND PAPER STRUCTURE

The primary contribution of this paper is a software framework that allows modular robot configurations and behaviors to be built hierarchically, and a simulation environment that helps the user verify intended behavior. Together, these tools help manage the complexity of a modular robot system, significantly reducing the time and effort required to accomplish tasks with modular robots.

The software we developed is open-source and freely available at **TODO: Link**. Our code is built for SMORES, a modular robot developed at the University of Pennsylvania (see sec 1), but could easily be adapted for use with other modular robot systems.

The remainder of this paper provides a comprehensive description of the structure and algorithmic components of our software system. In Section III, we discuss relevant background material. In Section IV we introduce terminology and concepts used elsewhere in the paper. In Section V, we describe the algorithmic basis for the three major components of our framework - design composition, behavior composition, and behavior verification. In Section VI, we discuss the open-source software tools used to implement our system, and provide examples demonstrating a user's workflow when using this system. We demonstrate that our framework saves the user time and effort, and allows him or her to easily develop complex and capable designs.

III. RELATED WORK

In some respect, our work parallels the efforts of Mehta [6] and Bezzo [1], who aim to create and program printable

robots from design specification by a novice user. Users create new designs by composing existing elements from a design library, and appropriate circuitry and control software are automatically generated as physical designs are assembled. The framework we present is intended specifically for modular robots, and consequently the workflow and design considerations are fundamentally different from that presented by Mehta and Bezzo. In traditional robot design (or printable robot design), hardware and software are somewhat decoupled - hardware is designed and built once, and then programmed many times. In the case of a MRS, the system can be reconfigured to meet new tasks, so hardware configuration and behavior programming go hand in hand. We intend our system to be fast enough that the user could conceivably develop and program a new design for every new task - designs are built once, and programmed once. Where Mehta et al provide many facilities to generate and verify low-level behaviors (i.e. motor drivers appropriate for motors, we do so for high-level behaviors.

A significant amount of work has been done in developing behaviors and software for modular robots. Much of this work focused on automatically generating designs and behaviors using artificial intelligence systems. Genetic algorithms have been applied for the automated generation of designs and behaviors [4]. Other work has focused on emergent behavior from distributed algorithms **TODO: cite papers**.

While significant progress has been made in the automated generation of modular robot behaviors, automated systems are not yet capable of making modular robots truly useful in practice [11]. The need for new programming techniques to manage the complexity of modular robot systems has been acknowledged in the literature [10]. Historically, gait tables have been a commonly used format in which open-loop kinematic behaviors can be easily encoded [9]. Phased automata have also been presented as a way to easily create scalable gaits for large numbers of modular robots [13]. In this paper, we present a novel scripting language to quickly create complex behaviors for modular robots.

Our framework assists users in verifying design validity by identifying self-collision and loss of gravitational stability. Identification of these conditions is common in modular robot reconfiguration planning [2] and motion planning [12].

IV. PRELIMINARIES

A. SMORES robot

We have developed our system for the SMORES modular robot, developed at the University of Pennsylvania [3]. Each SMORES module has four DoF (DoF) - three continuously rotating faces we call *turntables* and one central hinge with a 180° range of motion (Figure 1). The DoF marked 1, 2, and 4 have rotational axes that are parallel and coincident. **Each SMORES module can drive around as a two-wheel differential drive robot.** SMORES modules may connect to one another via magnets on each of their four faces, and are capable of self-reconfiguration.

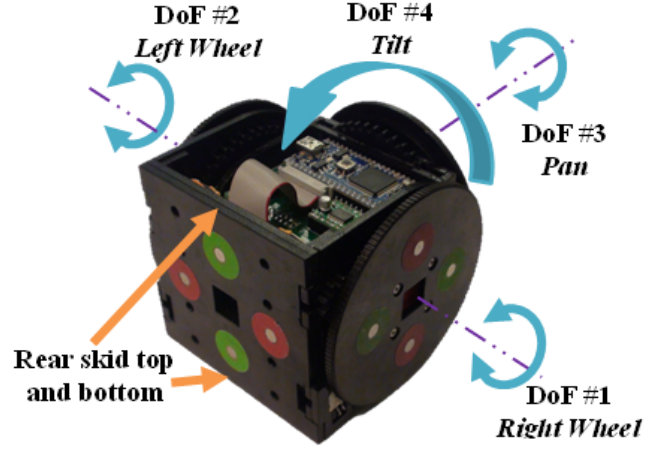


Fig. 1: SMORES robot

Note that while we demonstrate our software with SMORES, it is not limited to the SMORES robot - it could be applied to any modular robot which may be simulated using Gazebo.

B. Concepts and terms

A *configuration* is a contiguous set of connected modules which we treat as a single robot. Configurations are defined by their connective structure, and are represented by graphs with nodes representing modules and edges representing connections between modules. Edges are labeled to indicate which faces are connected as well as any angular offset, so that all information about the connective topology of the modules is captured. Individual modules are considered interchangeable (as long as they are of the same kind), so any two identically connected sets of modules are considered instances of the same configuration.

A *behavior* is a programmed sequence of movements for a specific configuration intended to produce a desired effect. A gait for walking is one example. In this paper, we consider open-loop kinematic behaviors represented as series-parallel action graphs, described in detail in section V-B.

A *controller* is a position and velocity servo for one DoF of a modular robot. A controller takes as input a desired position or angular velocity, and drives the error between the desired and actual state of the DoF it controls to zero over time.

When writing a behavior, it is possible to command one controller to simultaneously hold more than one desired position; this is known as a *controller conflict*. Behaviors with controller conflicts are impossible to execute.

During execution of a behavior, a *self-collision* can occur when two different parts of configuration are commanded to occupy the same location in space. Self-collisions can damage the robot, and are usually unwanted.

While executing many behaviors, it is desirable to maintain *gravitational stability* (also called quasi-static stability). Informally speaking, a robot is gravitationally stable when it is balanced, and gravity does not create any net moment on

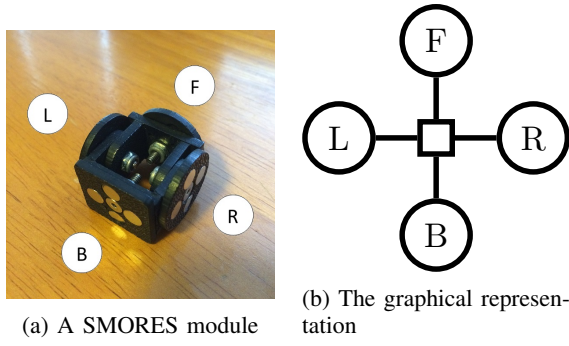


Fig. 2: A photo of a SMORES module and its graphical representation

it. Mathematically, the robot is gravitationally stable if the Z-projection of its center of mass lies within the convex hull of its load-supporting contact points in the ground plane.

V. APPROACH AND ALGORITHM

The three major components of our framework are configuration composition, behavior composition, and verification of configurations and behaviors. We begin with some definitions, and then address each of these three components.

Definition V.1 (Module). We define a single module as $\mathcal{M} = (P/\mathcal{R}, O/\mathcal{R}, V, N)$, where

- $P/\mathcal{R} = (p_x, p_y, p_z)$ is the position of the module in the global reference frame \mathcal{R} . We consider the position of the center of the module as the position of the module.
- $O/\mathcal{R} = (o_w, o_x, o_y, o_z)$ is the orientation of the module represented by quaternion in the global reference frame \mathcal{R} .
- $V = \{v_1, v_2, \dots, v_m\}$ is the set of joint values. Each joint value corresponds to each degree of freedom of the module.
- $N = \{n_1, n_2, \dots, n_k\}$ is the set of nodes where the module can connect to other modules.

For a single SMORES module, there are four degree of freedom as shown in Figure 1, and there are four nodes to connect to other modules, i.e. $N = \{\text{front, left, right, back}\}$ as shown in Figure 2.

Definition V.2 (Configuration). We define a configuration as $\mathcal{C} = (C, \gamma, M, \mathcal{M}_b, E, \delta)$, where

- C is a set of configurations, $C = \{C_1, C_2, \dots, C_q\}$.
- $\gamma : C \rightarrow M$ is a function that, when given a configuration, returns all modules of that configuration.
- M is the set of modules. If this configuration contains no other configurations, i.e. $C = \emptyset$, M is just the set of all modules of this configuration. If this configuration is composed by other configurations, i.e. $C = \{C_1, C_2, \dots, C_q\}$, we define $M = \bigcup_{C \in C} \gamma(C)$.
- E is a set of connections between modules. $(\mathcal{M}_i.n_i, \mathcal{M}_j.n_j) \in E$, where $\mathcal{M}_{i,j} \in M, \mathcal{M}_i \neq \mathcal{M}_j$, and $n_i \in \mathcal{M}_i.N, n_j \in \mathcal{M}_j.N$.

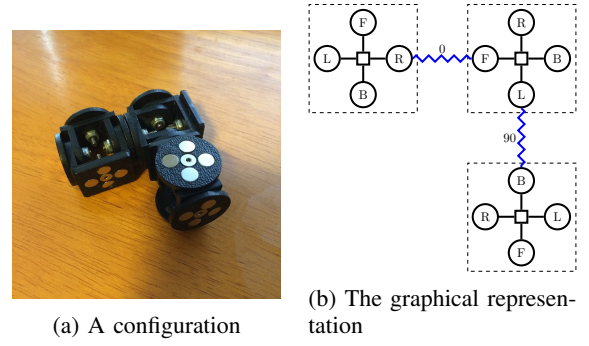


Fig. 3: A photo of a configuration with SMORES modules and its graphical representation

- $\mathcal{M}_b \in M$ is a single module that will be treated as a base module, i.e. $\nexists (\mathcal{M}_i.n_i, \mathcal{M}_j.n_j) \in E, s.t. \mathcal{M}_j = \mathcal{M}_b$.
- $\delta : E \rightarrow \mathbb{R}$ is a labeling function over a connection. Given a connection, δ returns the angle offset in degree between the two connected nodes.

We assume the set of connections is defined such that, if we form a directed graph by treating modules in M as nodes and connections in E as edges, the graph should not have any cycle. Figure 3a shows a photo of a configuration that does not include other configurations. Figure 3b shows its graphical representation. Blue zigzag lines represent connections between modules, and the label of each connection shows the angle offset of that connection. Notice that the definition of a configuration allow it to represent a complex structure with nested configurations with arbitrary depth.

A. Configuration Composition

Definition V.3 (Configuration Composition). Given a set of configurations C , a base configuration $C_b \in C$, and a set of connections E_C that encodes the connections between configurations in C , configuration composition combines all configuration in C to a single configuration C^* that has the following guarantees: i) C^* includes all modules and connections from C and E_C (section V-A1); ii) the positions of all modules in C^* are adjusted based on the connections to preserve structures of original configurations in C (section V-A2).

1) *Update Modules and Connections:* Before presenting the procedure to update modules and configurations in C^* , we will first discuss some assumptions about the set of connections E_C .

The set of connections E_C between configurations in C is defined as $(C_i.\mathcal{M}_i.n_i, C_j.\mathcal{M}_j.n_j) \in E_C$, where $C_{i,j} \in C, \mathcal{M}_i \in \gamma(C_i), \mathcal{M}_j \in \gamma(C_j)$, and $n_i \in \mathcal{M}_i.N, n_j \in \mathcal{M}_j.N$. Similar to the assumption about connections in a configuration, we assume that if we form a directed graph with configurations in C as nodes and connections in E_C as edges, the graph does not have any cycle.

Given a set of configurations C , a base configuration $C_b \in C$, and a set of connections E_C , we define the composed configurations to be $C^* = (C^*, \gamma, M, \mathcal{M}_b, E, \delta)$, where

- $C^* = C$
- $M = \bigcup_{C \in C^*} \gamma(C)$.
- $\mathcal{M}_b \in M$ is the base module of the base configuration \mathcal{C}_b .
- $E = (\bigcup_{(\mathcal{C}_i, \mathcal{M}_i, n_i, \mathcal{C}_j, \mathcal{M}_j, n_j) \in E_C} \{(\mathcal{M}_i, n_i, \mathcal{M}_j, n_j)\}) \cup (\bigcup_{C \in C^*} E \text{ of } C)$

The definitions of γ and δ are the same as the ones in Definition V.2.

2) *Update Positions of all Modules:* As multiple configurations are combined, positions and orientations of some modules in them need to be changed to preserve the relative positioning between modules as in the original configurations.

Algorithm 1 Update Module Position

Input:

A parent module \mathcal{M}_p
A child module \mathcal{M}_c
A connection $e = (\mathcal{M}_p, n_p, \mathcal{M}_c, n_c)$ and the labeling function δ

Output:

The child module \mathcal{M}_c with position and orientation updated

```

1: c_wrt_p ← get_transform( $\mathcal{M}_p.V_p, \mathcal{M}_p.V_p, e, \delta$ )
2: p_wrt_world ← get_matrix( $\mathcal{M}_p.P_p, \mathcal{M}_p.O_p$ )
3: c_wrt_world ← p_wrt_world · c_wrt_p
4:  $\mathcal{M}_c.P_c \leftarrow$  get_position(c_wrt_world)
5:  $\mathcal{M}_c.O_c \leftarrow$  get_orientation(c_wrt_world)
6: return  $\mathcal{M}_c$ 

```

Algorithm 2 Update All Module Positions

Input:

A configuration \mathcal{C}

Output:

The configuration \mathcal{C} whose module positions are all updated

```

1:  $\mathcal{M}_p \leftarrow \mathcal{C}.\mathcal{M}_b$ 
2:  $M = \{\mathcal{M}_p\}$ 
3:  $\mathcal{C}.M = \emptyset$ 
4: while  $M \neq \emptyset$  do
5:   for  $(\mathcal{M}_i, n_i, \mathcal{M}_j, n_j) \in \mathcal{C}.E$  do
6:     if  $\mathcal{M}_i = \mathcal{M}_p$  then
7:        $\mathcal{M}_j = \text{UpdateModulePosition}(\mathcal{M}_i, \mathcal{M}_j, (\mathcal{M}_i, n_i, \mathcal{M}_j, n_j), \mathcal{C}.\delta)$ 
8:        $M = M \cup \{\mathcal{M}_j\}$ 
9:        $\mathcal{C}.M = \mathcal{C}.M \cup \{\mathcal{M}_j\}$ 
10:    end if
11:  end for
12:   $M = M \setminus \{\mathcal{M}_p\}$ 
13:   $\mathcal{M}_p \leftarrow \text{RandomOneFrom}(M)$ 
14: end while
15: return  $\mathcal{M}_c$ 

```

Given a parent module \mathcal{M}_p , a child module \mathcal{M}_c , the connection between them and the corresponding labeling function, Algorithm 1 updates the position and orientation in the global reference frame of the child module based on the relative positioning between \mathcal{M}_p and \mathcal{M}_c . Function “get_transform” in line 1 computes the transformation matrix for the reference frame from the parent module to the child module. Function “get_matrix” in line 2 computes the transformation matrix from global frame to the parent module frame. With the two transformation matrices, line 3 computes the transformation matrix from the global frame to the child module frame. In line 4, 5, functions “get_position” and “get_orientation” update the position and orientation of the child module.

With the algorithm to update the position and orientation between two modules, Algorithm 2 shows the procedure to update the position and orientation of all modules in a given configurations \mathcal{C} . The algorithm starts with the base module of \mathcal{C} as shown in line 1. M is the set of modules who are updated already, but some modules that are directly connected to modules in M are not updated yet. The set of modules of \mathcal{C} is reset to empty set in line 3. The while loop from line 4 to 14 ensures all modules in \mathcal{C} are updated. The for loop and if statement in line 5, 6 find all connections in $\mathcal{C}.E$ whose first module is \mathcal{M}_p . In line 7 we apply Algorithm 1 to update the position and orientation of the second module in those connections. Line 8, 9 append the updated module into the set M and $\mathcal{C}.M$. In line 12, 13, we remove the current \mathcal{M}_p from the set M and randomly choose a new \mathcal{M}_p from M . Notice that the assumption we made about connections of a configuration in Definition V.2 prevents us from updating the same module twice.

B. Behavior Composition: Series-Parallel Action Graphs

We present a novel motion description language for modular robots. The language aims to balance simplicity and expressiveness, and is compositional in nature, designed to manage the complexity of developing complicated behaviors for large clusters of modular robots through abstraction and modularity. The language is in some ways similar to typical extended motion description languages (see [5]), but was specifically designed to facilitate composition.

The fundamental atoms of the language are called actions. An *action* is a tuple (J, X, ξ, T) , where J identifies a single DoF of a configuration, X is a controller setpoint for that degree of freedom, ξ specifies an interrupt condition, and T specifies a timeout. When an action executes, the controller setpoint (position or velocity) for the specified DoF is changed to the specified value. The controller maintains this setpoint until receiving a new one from another action.

The interrupt condition is a boolean function of the (sensed) state of the DoF J . When either the interrupt condition is met or time runs out (whichever comes first), the action is considered complete, and the next action begins. The interrupt condition can be set to *false* (so that only the timeout has effect), and T can be set to infinity (so that only the interrupt has effect).

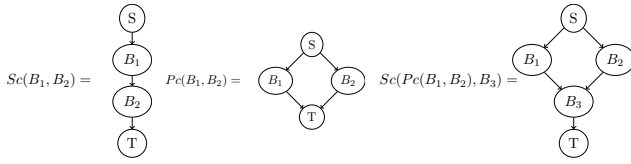


Fig. 4: Series and parallel composition of behaviors

Actions are composed to form behaviors. We define a *behavior* as a directed acyclic graph where nodes are actions and edges are transitions between actions. A behavior B always has two special nodes S and T , which are the *Start* and *Termination* nodes, respectively. The smallest behavior consists of S , T , and a single action. Behavior execution follows three simple rules:

- 1) Execution begins at S . S completes immediately.
- 2) Each action begins execution upon completion of *all* its parent actions.
- 3) All sequences of execution end at T .

Because execution begins at S and ends at T , there must be a (directed) path from every S to every node in B , and also from every node in B to T . Since B is acyclic, it is therefore a *directed series-parallel graph* (SPG). SPG's can always be formed recursively by parallel and series composition operations [8]. The parallel composition P of two behaviors B_1, B_2 , $P = Pc(B_1, B_2)$ is the disjoint union of their nodes (actions), merging S_1 with S_2 and T_1 with T_2 . The series composition of B_1, B_2 , $S = Sc(B_1, B_2)$ is created from their disjoint union by merging T_1 and S_2 , so that B_1 and B_2 execute sequentially¹. Note that if B_1 was itself created through parallel composition, B_2 will not begin until all chains of execution of B_1 are completed. Figure 4 provides a visual companion.

Example: A single SMORES module can use its left and right wheels to drive like a car. To drive forward, we might define a DRIVE behavior composing actions for the left and right wheels in parallel:

$$\text{DRIVE} = Pc \left(\begin{pmatrix} (L, \dot{\theta}_{set} = 6, \xi : false, T : 5), \\ (R, \dot{\theta}_{set} = 6, \xi : false, T : 5) \end{pmatrix} \right)$$

The wheels are set to turn at 6 radians per second, and the action will complete in 5 seconds. We might also define a TURN behavior, commanding the wheels to rotate π radians in opposite directions:

$$\text{TURN} = Pc \left(\begin{pmatrix} (L, \theta_{set} = \theta_0 + \pi, \xi : \theta == \theta_0 + \pi, T : \infty), \\ (R, \theta_{set} = \theta_0 - \pi, \xi : \theta == \theta_0 - \pi, T : \infty) \end{pmatrix} \right)$$

¹Since the merged node is not an action, we can freely omit it and instead draw edges from each of its parent nodes to each of its child nodes.

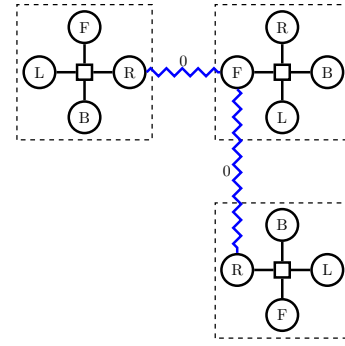


Fig. 5: A configuration with self-collision

Here, θ_0 denotes the currently-sensed value of theta at the beginning of the TURN behavior. The action completes when both wheels actually reach their commanded angles of $\theta_0 \pm \pi$. To drive in a square, we compose DRIVE and TURN behaviors in series:

$$\text{SQUARE} = Sc(\text{DRIVE}, \text{TURN}, \text{DRIVE}, \text{TURN}, \text{DRIVE}, \text{TURN}, \text{DRIVE}, \text{TURN})$$

C. Verification of Configuration and Control

1) Verification of Configurations: In section V-A, we introduced algorithms to compose a set of configurations into a single configuration. However, it might not be possible or safe to form the structure represented by the composed configuration with the actual module. Consider the configuration shown in Figure 5. It is easy to tell that such configuration is impossible to build with modules that allow only one connection at each node. Thus it is important to verify whether the configuration is valid or not for a given modular robot system.

Definition V.4 (Verification on Configuration). Given a configuration \mathcal{C} , we say \mathcal{C} is valid if it satisfied the following set of properties

- There is no collision between modules in the configuration.
- The configuration is statically stable.

Notice that in order to verify the validity of the configuration, one needs to know some properties of all modules for the given modular robot system, e.g. the geometry information, the mass of each module.

a) Collision Checking: With positions and orientations of all modules in the configuration update, and the geometry information of all module, we can check whether any two modules occupy the same space. If so, there exist a self-collision in the configuration.

b) Stability Checking: The static stability is checked by first computing the location of the center of mass of the configuration. For the given configuration \mathcal{C} , the position of

the center of mass is

$$P_C = \frac{\sum_{M \in \gamma(C)} M.P \cdot M_m}{\sum_{M \in \gamma(C)} M_m}$$

where M_m is the mass of the module M .

Then we find the set of modules M_c that have minimal position in the z direction and consider them as the contact points between the ground plane and the configuration. A set of points in the $x-y$ plane can be extracted from M_c as $\sigma = \{(M.P.x, M.P.y) \mid M \in M_c\}$. If point $(P_C.x, P_C.y)$ is inside of the convex polygon formed by the set of points σ , the given configuration C is considered statically stable.

2) *Verification of Control*: In section V-B, we introduced a novel motion description language for modular robots. Actions defined by the language can be combined to produce more complex behaviors. Similar to configuration composition, we want to make sure the composed behaviors are valid and safe to execute. For example, a behavior that results two module colliding with each other during the execution should be considered unsafe. As the composed behaviors are used to control modular robots to produce locomotion or actions, we also refer to a behavior as a controller.

Definition V.5 (Verification on Controller). A controller is valid if it satisfied the following set of properties when controlling a configuration of a given modular robot system

- There is no collision between modules at all time during the execution of the controller.
- The configuration is statically stable at the end of the execution.
- The maximum duration when the configuration is not statically stable during the execution is less than a time bound t .

Similar to the verification of a configuration, we need to know the geometry information and the mass of each module for the given modular robot system.

D. Complexity

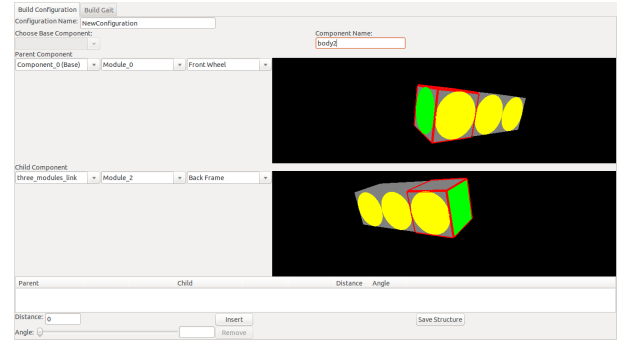
Discuss the complexity of the algorithm with respect to the number of modules and size of gait tables.

VI. IMPLEMENTATION

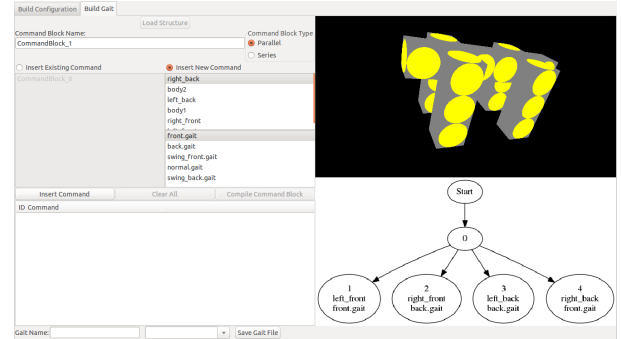
We implement a program to aid users to design and verify complex configurations and controllers from a set of basic configurations and associated controllers. We separated the program into two main parts, a configuration builder and a controller builder.

A. Configuration Builder

Given a set of basic configurations, the configuration builder allows users to combine basic configurations by choosing connection node on each configuration, as demonstrated by green nodes in Figure 6a. In addition, the configuration builder will also warn users when the composed configuration is not valid without the usage of a physical simulator, e.g. Gazebo.



(a) GUI for configuration builder



(b) GUI for controller builder

Fig. 6: The program to design and verify configurations and controllers

B. Control Builder

Given a composed configuration from the configuration builder, the controller builder aids user to design controllers for the composed configuration by arranging a set of basic controllers in parallel or in series. Figure 6b illustrates a new controller is composed by putting four basic controllers in parallel. Similar to the configuration builder, the controller builder will also warn users if there is self-collision in the configuration during the execution of composed controllers without simulations in a physical engine.

TODO: Tarik and Jim write

With simulation in Gazebo:

- Show a configuration composed from a set of basic configurations.
- Show a composed controller that results in a collision in the configuration.
- Show an updated controller that resolves the collision
- Show a composed controller that results in an unexpected behavior.
- Show an updated controller that eliminates the unexpected behavior.

VII. EXAMPLES

Here we present some examples to illustrate important features of our framework.

A. Toward a Standard Library

Our eventual intention is to develop a large library of configurations and associated behaviors which are available to all users of our framework, analogous to the standard libraries of major programming languages. The compositional nature of our framework will allow users to rely heavily on the library when approaching new tasks, allowing them to create sophisticated robots very quickly.

As a first step toward a standard library, we present a small library of configurations and associated behaviors in Tables I and II. Configurations in the library are organized by *order*, defined recursively as follows: a single module is an order-zero configuration, and the order of all other configurations is one greater than the largest order of the sub-configurations from which it is composed. Associated with each configuration is a set of behaviors. **TODO: Behaviors are grouped by functionality?**

For the library to be most effective, the set of configurations and behaviors available at each level (and especially at the lowest levels) should provide a rich set of functionalities without presenting the user with an overwhelming number of options. Considering the small library in Tables I and II, it is interesting to note that a large and diverse set of second- and third-order configurations can be constructed from only three first-order configurations. Developing metrics to evaluate the quality of such a library is an interesting opportunity for future work.

B. The User Perspective

We present the start-to-end user perspective in designing a complicated configuration called Walkbot. Consider a basic configuration formed by three modules in a line. We can form a “body” configuration by connecting two of the basic configurations as shown in Figure 7a. With four more basic configurations and attach sides of those basic configurations to sides of the “body” configuration with certain angle offset, we can build a complex configuration, Walkbot, with four legs, as shown in Figure 7b. Notice that by connecting multiple basic configurations together, we can design a complex configuration without building with individual modules.

C. Easy scale-up through composition

Our framework allows users to quickly create and program large configurations. The first order CHAIN3 configuration (Figure ??) can use a *sineGait* behavior to locomote like a snake. Defining *sineGait* as the series composition of two half-waves will allow us to re-use the gait with larger snakes:

$$\text{sineGait} = \text{Sc}(\text{halfSine1}, \text{halfSine2})$$

Arbitrarily long snake configurations can be created by composing CHAIN3’s end-to-end; Figure ?? shows one with 18 modules. A gait for an arbitrarily long snake is created by composing sine wave gaits for each component in parallel,

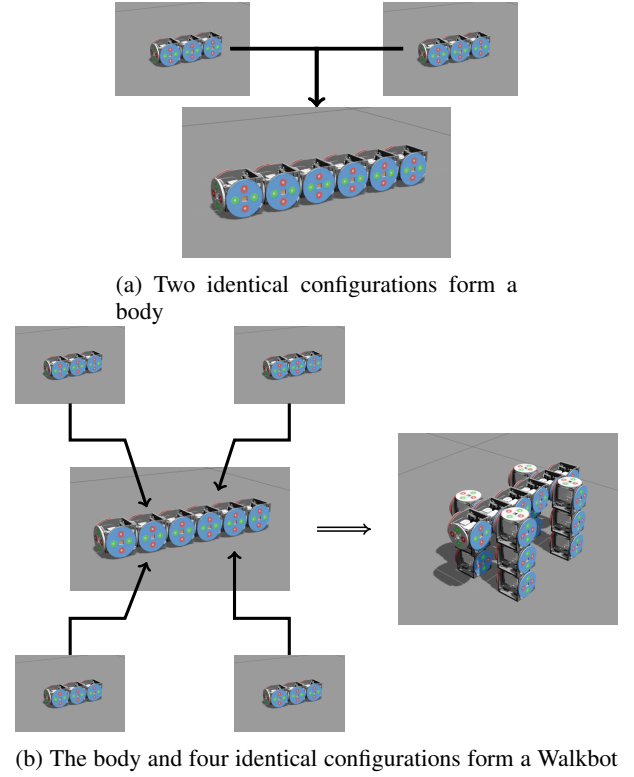


Fig. 7: Building a Walkbot with six identical configurations

but with alternating phase:

$$n\text{SnakeSineGait} = P_c \left(\text{Sc} \left(\begin{matrix} \text{halfSine1} \\ \text{halfSine2} \end{matrix} \right), \text{Sc} \left(\begin{matrix} \text{halfSine2} \\ \text{halfSine1} \end{matrix} \right), \dots \right)$$

D. Verification

The need for verification becomes more important as design complexity increases. Consider the Backhoe design, composed of the 2nd-order car and PUMA arm configurations. It is easy for this design to become gravitationally unstable. Our verification system warns the user when this happens.

VIII. RESULTS

In the past, designing configurations and behaviors to address new tasks has required time on the order of one day [7]. Using our framework and library, complex designs (such as the Walkbot or 18-module snake) can be created and programmed in under an hour. **TODO: Mark, what is the best way to present this comparison?**

IX. CONCLUSIONS

We worked hard, and had fun.

X. FUTURE

- How to represent different attribute/ability of the configurations

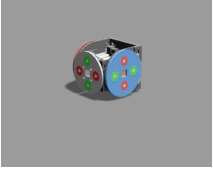
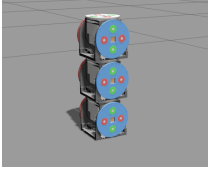
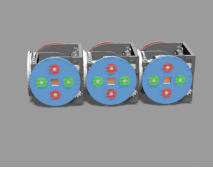
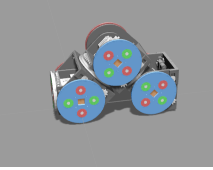
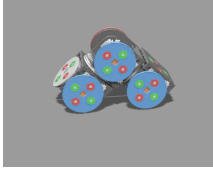
Config- uration					
Behaviors	$Drive(v, t)$ $TiltMiddleUp()$	$Step()$	$HoldRigid()$	$Steer(\theta)$ $DriveSideWheels(v, t)$	$Wiggle()$ $AntiWiggle()$

TABLE I: Tier-1 configurations (basic configurations)

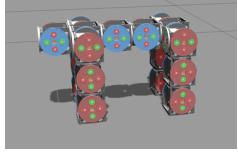
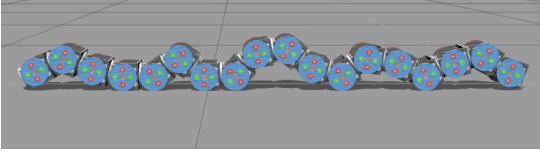
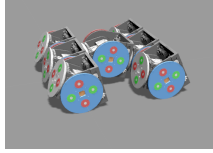
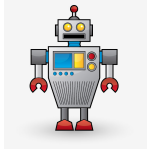
Config- uration				
Components	BODY3 LEG3 $\times 4$	SNAKE3 $\times 6$	STEER3 DRIVER1 $\times 4$ $DRIVE(v, t)$ $TURN(\theta)$	STEER3 DRIVER1 $\times 2$ $DRIVE(v, t)$ $TURN(\theta)$
Behaviors	$Walk(t)$	$Slither()$		

TABLE II: Tier-2 configurations (composed of basic configurations)

REFERENCES

- [1] Nicola Bezzo, Junkil Park, Andrew King, Peter Gebhard, Radoslav Ivanov, and Insup Lee. Demo abstract: Roslaba modular programming environment for robotic applications. In *Cyber-Physical Systems (ICCPs), 2014 ACM/IEEE International Conference on*, pages 214–214. IEEE, 2014.
- [2] Aranzazu Casal. *Reconfiguration planning for modular self-reconfigurable robots*. PhD thesis, Stanford University, 2001.
- [3] J. Davey et al. Emulating self-reconfigurable robots: Design of the smores system. In *Proc. IEEE Intelligent Robots Systems Conf.*, 2012.
- [4] Gregory S Hornby, Hod Lipson, and Jordan B Pollack. Generative representations for the automated design of modular physical robots. *Robotics and Automation, IEEE Transactions on*, 19(4):703–719, 2003.
- [5] D Hristu-Varsakelis, S Anderson, F Zhang, P Sodre, and PS Krishnaprasad. A motion description language for hybrid system programming. *Institute for Systems Research (preprint)*, 2003.
- [6] Ankur Mehta, Nicola Bezzo, Peter Gebhard, Byoungkwon An, Vijay Kumar, Insup Lee, and Daniela Rus. A design environment for the rapid specification and fabrication of printable robots. In *International Symposium on Experimental Robotics (ISER), Marrakech, Morocco*, 2014.
- [7] Jimmy Sastra. Using reconfigurable modular robots for rapid development of dynamic locomotion experiments. 2011.
- [8] Jacobo Valdes, Robert E Tarjan, and Eugene L Lawler. The recognition of series parallel digraphs. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12. ACM, 1979.
- [9] M. Yim. *Locomotion with a unit-modular reconfigurable robot*. PhD thesis, Stanford, 1994.
- [10] Mark Yim, David G Duff, and Kimon Roufas. Modular reconfigurable robots, an approach to urban search and rescue. In *1st Intl. Workshop on Human-friendly Welfare Robotics Systems*, pages 69–76, 2000.
- [11] Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *Robotics & Automation Magazine, IEEE*, 14(1):43–52, 2007.
- [12] Eiichi Yoshida, Satoshi Matura, Akiya Kamimura, Kohji Tomita, Haruhisa Kurokawa, and Shigeru Kokaji. A self-reconfigurable modular robot: Reconfiguration planning and experiments. *The International Journal of Robotics Research*, 21(10-11):903–915, 2002.
- [13] Ying Zhang, Mark Yim, Craig Eldershaw, Dave Duff, and Kimon Roufas. Phase automata: a programming model of locomotion gaits for scalable chain-type modular robots. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2442–2447. IEEE, 2003.