Computer-aided Compositional Design and Verification for Modular Robots

Author Names Omitted for Anonymous Review. Paper-ID [168]

Abstract To take full advantage of the flexibility of a modular robot system, users must be able to create new configurations and behaviors quickly. We present a design framework that facilitates rapid creation of new configurations and behaviors through composition of existing ones, and tools to verify configurations and behaviors as they are being created. New configurations are created by combining existing sub-configurations, for example combining four legs and a body to create a walking robot. Behaviors are associated with each configuration, so that when sub-configurations are composed, their associated behaviors are immediately available for composition as well. We introduce a new motion description language (Series-Parallel Action Graphs) that facilitates the rapid creation of complex behaviors by composition of simpler behaviors. We provide tools that automatically verify configurations and behaviors during the design process, allowing the user to identify problems early and iterate quickly. After verification, users can evaluate their configurations and behaviors in a physical simulator. The software we have developed is open-source, and will be made freely available online.

1 Introduction

Modular reconfigurable robot systems have been studied extensively for several decades. These systems distinguish themselves from conventional robotic systems in their ability to transform into different shapes to address a wide variety of tasks. They promise to be versatile, robust, and low cost [23]. Dozens of groups have different kinds of reconfigurable robots [5, 11], and introduced approaches for programming them [17, 20, 14]. Over 800 papers, a book [9], and a survey [24] have been written on the subject.

This versatility places an additional burden on the user, because solving problems with modular robots involves not only designing programs, but also the best physical form for the task at hand. If this complexity is not appropriately managed, it will present a significant barrier to using modular robots to address practical tasks [22]. If the user is free to create any new design 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 versatile modular hardware will be defeated.

Software modularity is a well-established practice for developing large maintainable systems and avoiding duplication of effort. In robotics, software behaviors are inextricably linked to the hardware they control, resulting in challenges to making modularity effective. Strong ties to physical hardware also increase the need for software verification, to ensure that hardware is not damaged. Significant progress has been made on these fronts in traditional robotics, most notably ROS [16] which provides inter-process communication and standard libraries for common robot tasks, as well as powerful verification tools [8].

In modular robotics, the challenge is different. Modular robot systems are not optimized for specific tasks, so in order for them to be useful, we must take full advantage of their flexibility. To do so, a user must be able to generate and verify configurations and behaviors as quickly as possible.

Toward that end, we present a design framework that facilitates the rapid creation of new configurations and behaviors through composition, and tools to verify configurations and behaviors while they are being created. New configurations are created by combining existing sub-configurations, for example combining four legs and a body to create a walking robot. Behaviors are associated with each configuration, so that when sub-configurations are composed, their associated behaviors are immediately available for composition as well. We introduce a new motion description language (Series-Parallel Action Graphs) that facilitates the rapid creation of complex behaviors by composition of simpler behaviors. We provide tools that automatically verify configurations and behaviors during the design process, allowing the user to identify problems early and iterate quickly. After verification, users can evaluate their configurations and behaviors in a physical simulator. The software we have developed is open-source, and will be made freely available online.

The remainder of this paper provides a description of the structure and algorithmic components of our framework. In Section 2, we discuss relevant background material. In Section 3 we introduce terminology and concepts used elsewhere in the paper. In Section 4, we describe the algorithmic basis for the three major components of our framework - design composition, behavior composition, and verification. In Section 5, we discuss the open-source software tools used to implement our framework. In Section 6, we provide examples highlighting important aspects of the framework, including a demonstration of the user's workflow. We demonstrate that our framework saves the user time and effort, and allows him or her to easily develop complex and capable designs.

2 Related Work

In some respects, our work parallels the efforts of Mehta [13] and Bezzo [1], who aim to create and program printable robots from novice users' design specifications. 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 modular robot system, 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 (*e.g.* 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 [6]. Other work has focused on distributed control [20] and hormone-based control [17].

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 [24]. The need for new programming techniques to manage the complexity of modular robot systems has been acknowledged in the literature [22]. Historically, gait tables have been a commonly used format in which open-loop kinematic behaviors can be easily encoded [21]. Phased automata have also been presented as a way to easily create scalable gaits for large numbers of modular robots [26]. In this paper, we introduce a novel motion description language that enables users to quickly create behaviors for modular robots.

Our framework assists users in verifying design validity by identifying self-collision and loss of gravitational stability. In existing literature, these conditions have been checked in the context of modular robot reconfiguration planning [3] and motion planning [25]. To our knowledge, there is no modular robot design tool that verifies these conditions to provide assistance to a human designer.

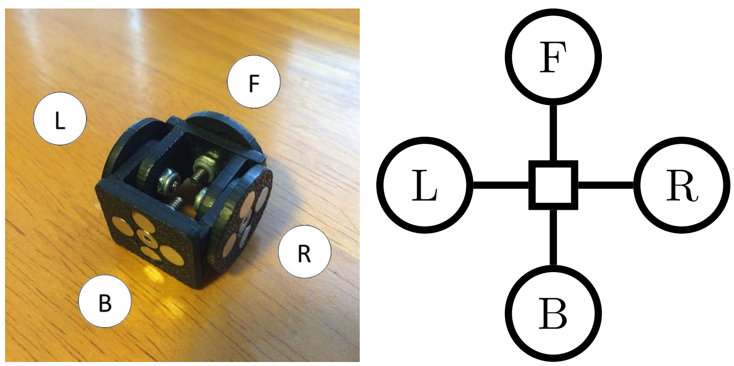
3 Definitions

In this section, we present concepts and terms which will be used later in the paper.

Definition 1 (Module). A module is a small robot that can move, respond to commands, and attach to other modules. Formally, we define a module as $\mathscr{M} = ({}^{\mathscr{W}}D^{\mathscr{M}}, X, A, K)$, where:

- ${}^{\mathscr{W}}D^{\mathscr{M}} \in SE(3)$ is the rigid-body *displacement* (position and orientation) of the module body frame in the world reference frame $\mathscr{W}$.
- $X = \{x_1, x_2, \ldots, x_d\}$ is the *state* of the module, with each x_i corresponding to one of the d degrees of freedom (DoF) of the module.
- $A = \{a_1, a_2, ..., a_k\}$ is the set of *attachment points* where the module can connect to other modules.
- $K : (X, a_i) \rightarrow SE(3)$ is the module's *forward kinematics function*, returning ${}^{\mathscr{B}}D^{a_i}$ (the displacement of attachment point a_i in the body frame) as a function of X

Figure 1 shows a schematic representation of a module with four attachment points



(a) A module (b) The graphical representation

Fig. 1: A photo of a module and its graphical representation

Definition 2 (Configuration). A *configuration* is a contiguous set of connected modules which we treat as a single robot. The identity of a configuration is determined by its connective structure; configurations can be represented by graphs with nodes representing modules and edges representing connections between modules. Individual modules are considered interchangeable (as long as they are of the same kind).

In this paper, we present an object-oriented design framework for modular robot systems, and treat configurations as the fundamental objects. Rather than defining configurations only by the topology of their component modules, we define them recursively, as being composed of connected sub-configurations. A single module is considered the smallest configuration.

Formally, we define a configuration as $\mathscr{C} = (C, \gamma, M, E, \delta, X, B)$, where:

- C is a set of sub-configurations, $C = \{\mathscr{C}_1, \mathscr{C}_2, ..., \mathscr{C}_q\}$.
- $\gamma : C \to 2^M$ is a function mapping a configuration $\mathscr{C} \in C$ to its set of modules.
- $M = \bigcup_{\mathscr{C} \in C} \gamma(\mathscr{C})$ is the set of modules.
- E is a set of connections between modules. $(\mathscr{M}_i.a_i, \mathscr{M}_j.a_j) \in E$, where $\mathscr{M}_i, \mathscr{M}_j \in M, \mathscr{M}_i \neq \mathscr{M}_j$, and $a_i \in \mathscr{M}_i.A$, $a_j \in \mathscr{M}_j.A$.
- $\delta : E \to SO(3)$ is a labeling function over connections returning ${}^{\mathscr{M}_i.a_i}R^{\mathscr{M}_j.a_j}$, the orientation of one attachment point relative the other.
- $X = \bigcup\limits_{\mathscr{M}_i \in M} \mathscr{M}_i.X$ is the *state* of the configuration.
- B is a set of *behaviors* (Definition 3) associated with the configuration.

Figure 2a shows a photo of a configuration composed of three modules, each with four attachment points. Figure 2b shows its graphical representation. Blue zigzag lines represent connections between modules, and the label of each connection shows the angle offset of that connection.

In this paper, we consider only configurations without cycles. While this is a constraint, the set of functionalities provided by acyclic designs is very rich, so we feel that it does not unreasonably limit the utility of our framework. Accommodating designs with cycles is left to future work.

Assuming acyclic configurations, we can compute forward kinematics for the entire configuration by composing displacements module-to-module. Let any module $\mathscr{M}_f \in M$ have fixed displacement ${}^{\mathscr{W}}D^{\mathscr{B}_f}$ in the world frame. Let $\mathscr{M}_i : (\mathscr{M}_i.a_i,\ \mathscr{M}_f.a_f) \in E$ be connected to $\mathscr{M}_f$. We can find ${}^{\mathscr{W}}D^{\mathscr{M}_i}$ by composing displacements as follows:

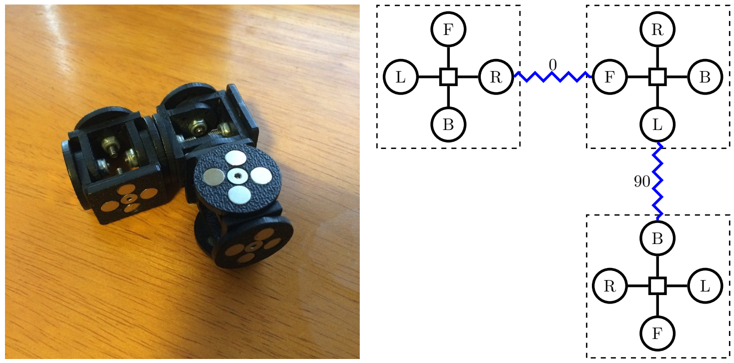
$$\begin{aligned} {}^{\mathscr{W}}D^{\mathscr{M}_i} &= [{}^{\mathscr{W}}D^{\mathscr{M}_f}][{}^{\mathscr{M}_f}D^{a_f}][{}^{a_f}D^{a_i}][{}^{\mathscr{M}_i}D^{a_i}]^T \\ &= [{}^{\mathscr{W}}D^{\mathscr{M}_f}][K_f(X_f, a_f)] \begin{bmatrix} \delta(e) & 0 \\ 0 & 1 \end{bmatrix} [K_i(X_i, a_i)]^T \end{aligned}$$

where $e = (\mathscr{M}_i.a_i,\ \mathscr{M}_f.a_f)$. To find the world-frame displacements of all other modules, we may traverse the connections of the configuration, repeatedly composing displacements in the manner above.

Definition 3 (Behavior). 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 4.2.

Definition 4 (Controller). A *controller* is a position or 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.

Definition 5 (Self-Collision). 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.



(a) A configuration (b) The graphical representation

Fig. 2: A photo of a configuration with three modules and its graphical representation

Definition 6 (Gravitational Stability). 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 it. Mathematically, the robot is gravitationally stable if the projection of its center of mass onto the group plane lies within the convex hull of its load-supporting contact points in the ground plane.

4 Approach and Algorithm

The three major components of our framework are configuration composition, behavior composition, and verification of configurations and behaviors.

4.1 Configuration Composition

Before discussing about configuration composition, we will first define a set of connections E_C between configurations in a given set C as $(\mathscr{C}_i.\mathscr{M}_i.a_i, \mathscr{C}_j.\mathscr{M}_j.a_j) \in E_C$, where $\mathscr{C}_i, \mathscr{C}_j \in C$, $\mathscr{M}_i \in \gamma(\mathscr{C}_i), \mathscr{M}_j \in \gamma(\mathscr{C}_j)$, and $a_i \in \mathscr{M}_i.A$, $a_j \in \mathscr{M}_j.A$. Similar to the assumption about connections in a configuration, we assume that we form an acyclic graph with configurations in C as nodes and connections in E_C as edges.

Definition 7 (Configuration Composition). Given a set of configurations C and a set E_C of connections between them, configuration composition combines all configurations in C to a single configuration $\mathscr{C}^*$ that includes all modules and connections from C and E_C.

Given a set of configurations C and a set of connections E_C, we define the composed configuration to be $\mathscr{C}^* = (C^*, \gamma, M, E, \delta, X, B)$, where

- $C^* = C$
- $M = \bigcup_{\mathscr{C} \in C^*} \gamma(\mathscr{C})$.
- $E = (\bigcup_{(\mathscr{C}_i.\mathscr{M}_i.n_i, \mathscr{C}_j.\mathscr{M}_j.n_j) \in E_C} \{(\mathscr{M}_i.n_i, \mathscr{M}_j.n_j)\}) \cup (\bigcup_{\mathscr{C}_i \in C^*} E_i)$
- $B = \bigcup_{\mathscr{C}_i \in C} \mathscr{C}_i.B$

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

4.2 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 designed specifically for composition, allowing complex behaviors to be rapidly created by combining existing behaviors. Typical motion description languages, which have atomic elements that represent controller commands (set-points and gains) with limited duration [2]. Extended motion description languages introduce interrupts to control the duration of commands [7]. Our language distinguishes itself by allowing existing behaviors to be easily composed to form new behaviors.

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 DoF, 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). As an example, the action ($Module0\_L$, $\theta_{set} = \pi$, $\xi : \theta == \pi$, $T : \infty$) encodes "Command the controller of the left wheel of module zero to maintain a setpoint of π radians. When the encoder of that wheel indicates that π radians has actually been reached, consider this action complete and move on."

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 [19]. 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 sequentially[1]. 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 3 provides a visual companion.

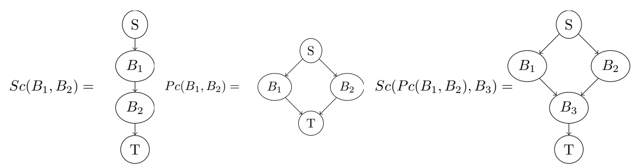


Fig. 3: Series and parallel composition of behaviors

Example

Consider a single module that has two wheels that allow it to drive like a car. To drive forward, we might define a DRIVE behavior composing actions for the left and right wheels in parallel:

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

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:

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

[1] Since 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.

Here, θ_0 denotes the currently-sensed value of θ 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:

$$\begin{aligned}\text{SQUARE} = Sc(\ &\text{DRIVE},\ \text{TURN},\ \text{DRIVE},\ \text{TURN},\\ &\text{DRIVE},\ \text{TURN},\ \text{DRIVE},\ \text{TURN})\end{aligned}$$

4.3 Verification of Configuration and Behavior

4.3.1 Verification of Configurations

In section 4.1, we introduced the definition on composing 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 4. It is easy to tell that such configuration is not ideal for any task, because the self-collision may damage those modules and result in instability of the system. Thus it is important to verify whether the configuration is valid or not for a given modular robot system.

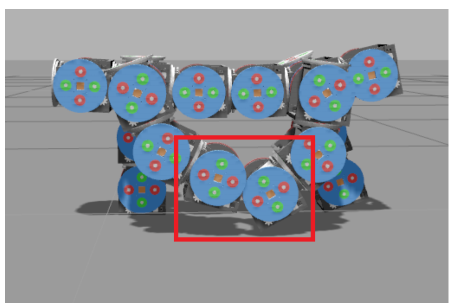


Fig. 4: A configuration with self-collision

Definition 8 (Configuration Verification). Given a configuration $\mathscr{C}$ and state X_0, we say $\mathscr{C}$ is valid in state X_0 if it satisfied the following set of properties:

- There is no collision between modules in the configuration.
- The configuration is gravitationally 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, and the mass of each module.

Collision Checking

We obtain the positions and orientation of all modules through forward kinematics. Using the known model geometry, we check whether any two modules occupy the same space. If so, there exists a self-collision in the configuration. Our implementation checks self-collision by approximating modules as spheres, and checking the distance (radius) between all pairs. More sophisticated techniques are available which efficiently produce exact results [15].

Stability Checking

Gravitational stability is checked by computing the location of the center of mass of the configuration. For the given configuration $\mathscr{C}$ and state X_0, the position of the center of mass is

$$P_{\mathscr{C}} = \frac{\sum\limits_{\mathscr{M} \in \gamma(\mathscr{C})} P(\mathscr{M}) \cdot \mathscr{M}_m}{\sum\limits_{\mathscr{M} \in \gamma(\mathscr{C})} \mathscr{M}_m}$$

where $\mathscr{M}_m$ is the mass of the module $\mathscr{M}$, and $P(\mathscr{M})$ is a function returning the position coordinates of $\mathscr{M}$ in the world frame.

We find the set of modules M_c that have minimal position in the z direction and consider them to be the contact points between the ground plane and the configuration. We treat the set $\sigma = P(\mathscr{M}), \forall \mathscr{M} \in M_c$ as an approximate set of contact points. If the projection of P_c onto the ground plane lies within the convex hull form by the set of points in σ, configuration $\mathscr{C}$ is considered statically stable in state X_0.

4.3.2 Verification of Behavior

In section 4.2, 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 in two modules colliding during the execution should be considered unsafe.

Definition 9 (Behavior Verification). A behavior is valid if it satisfies the following set of properties when controlling a configuration of a given modular robot system

- There are no collisions between modules at all times during the execution of the controller.
- There are no behavior conflicts at all times during the execution.
- The configuration is gravitationally stable at the end of the execution.
- The maximum duration for which the configuration is not gravitationally stable during the execution is less than a time bound t_{max}.

For a behavior with duration time T_B, we check the state of the configuration with sampling time $t_B < t_{max}$. For each sample, we first detect behavior conflict by checking if different commands are given to the same joint of a module simultaneously. If there is no behavior conflict, we update the positions and orientations of all modules in the configuration based on behavior commands. Then we check collision and gravitational stability of each configuration as discussed in section 4.3.1. We argue that this behavior is not safe, if i) there are n consecutive samples when the configuration is not gravitationally stable and $n \cdot t_B > t_{max}$; or ii) the configuration at time T_B is not gravitationally stable.

5 Implementation

We implement a design interface to aid users to design and verify complex configurations and behaviors from a set of basic configurations and associated behaviors. We separated the program into two main parts, a configuration builder and a behavior builder. Our implementation is built for the SMORES modular robot, but could easily be adapted to other modular robot systems.

5.1 SMORES robot

We have developed our system for the SMORES modular robot, developed at the University of Pennsylvania [4]. Each SMORES modules has four DoF - three continuously rotating faces called *turntables* and one central hinge with a 180° range of motion (Figure 5). 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. Formally, we denote the state of a SMORES module as $X = \{\theta_L, \theta_R, \theta_F, \theta_B\}$ and the set of attachment points as $A = \{L, R, T, B\}$.

5.2 Configuration Builder

Given a set of basic configurations which could be just single modules, 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 warn users when the composed configuration is not valid without the usage of a physical simulator, e.g. Gazebo [10].

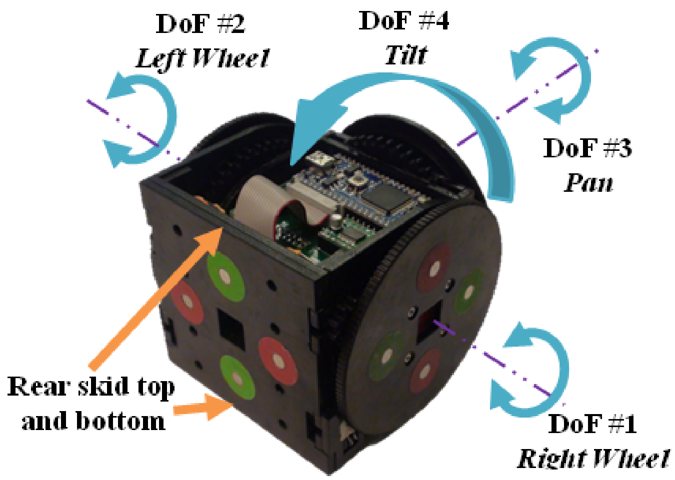


Fig. 5: SMORES robot

5.3 Behavior Builder

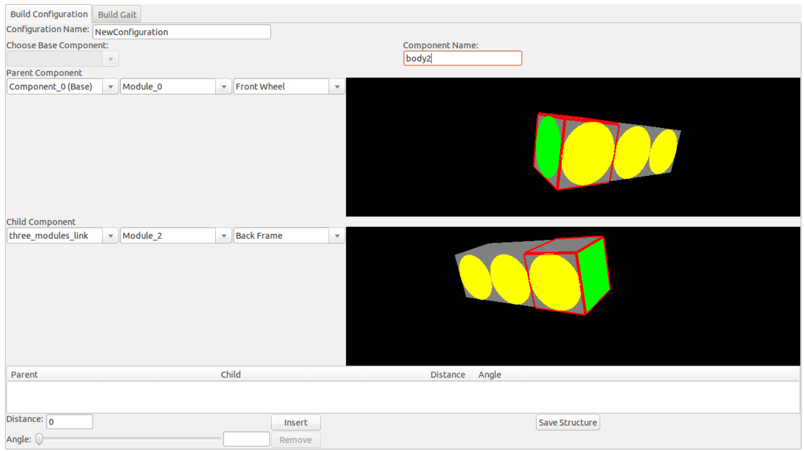
Given a composed configuration, the behavior builder aids users in designing behaviors for the composed configuration by arranging a set of basic behaviors in parallel or in series. Figure 6b illustrates a new behavior is composed by putting four basic behaviors in parallel. Similar to the configuration builder, the behavior builder will also warn users if there are self-collisions in the configuration during the execution of composed behaviors without simulations in a physical engine.

6 Examples

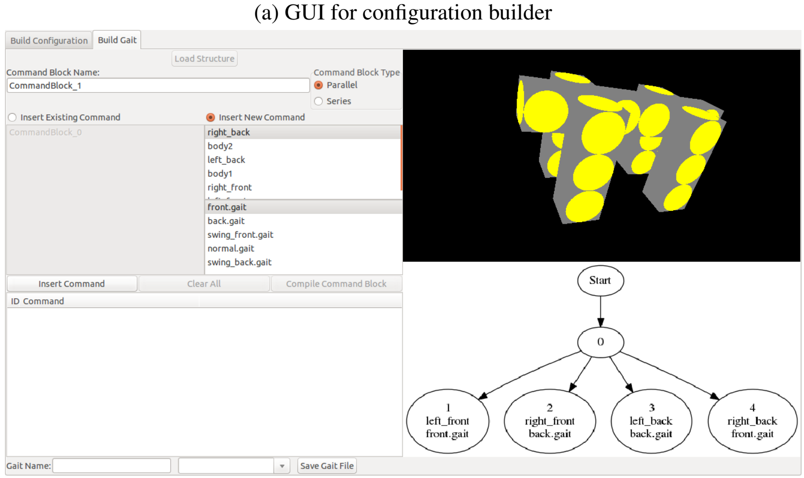
Here we present a few examples to illustrate important features of our framework.

6.1 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.



(a) GUI for configuration builder



(b) GUI for behavior builder

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

As a first step toward a standard library, we present a small library of configurations and associated behaviors in Tables 1 and 2. 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. Each configuration has an associated set of behaviors, which the user can compose to accomplish tasks. New behaviors for higher-order configuration can be created by composing the behaviors of its component sub-configurations.

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 1 and 2, it is interested to note that a diverse set of second- and third-order configurations can be constructed from only one zero- and one first-order configuration. Developing metrics to evaluate the quality of such a library is an interesting opportunity for future work.

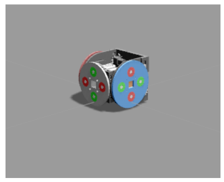
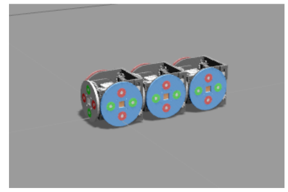
	Order-0 (single module)	Order-1
Configuration	 MODULE	 CHAIN3
Behaviors	$Drive(v,t)$, $TiltMiddle(\theta)$, $SpinTop(\theta)$	$Drive(v,t)$, $SteeringPose()$, $LegStep()$, $HoldRigid()$, $Steer(\theta)$

Table 1: Order-0 and Order-1 configurations

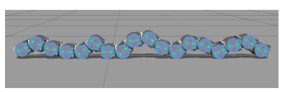
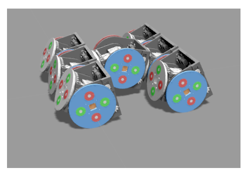
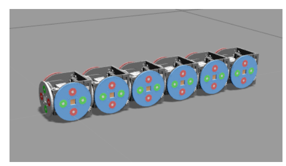
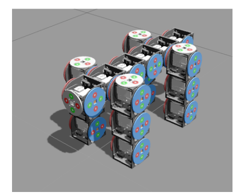
	Order-2	Order-2	Order-2	Order-3
Configuration	 CHAIN18	 CAR	 CHAIN6	 WALKBOT
Components	CHAIN3 $\times 6$	CHAIN3 MODULE $\times 4$	CHAIN3 $\times 2$	CHAIN6 CHAIN3 $\times 4$
Behaviors	$SineGait18()$	$Drive(v,t)$ $Turn(\theta)$	$Drive(v,t)$, $Turn(\theta)$ $HoldRigid()$	$Walk()$

Table 2: Order-2 and Order-3 configurations

6.2 The User Perspective

Figure 7 demonstrates the design flow when a user is designing a configuration an its behaviors. We present the start-to-end user perspective in designing a complicated configuration called Walkbot. Consider a basic configuration “CHAIN3” as shown in Table 1. We can form a “CHAIN6” configuration, as a body, by connecting two of the basic configurations as demonstrated in Figure 8a. With four more basic configurations attaching to sides of the “CHAIN6” configuration with certain angle offset, we can build a complex configuration, Walkbot, with four legs, as shown in Figure 8b. Notice that by connecting multiple basic configurations together, we can design a complex configuration more easily than creating with individual modules.

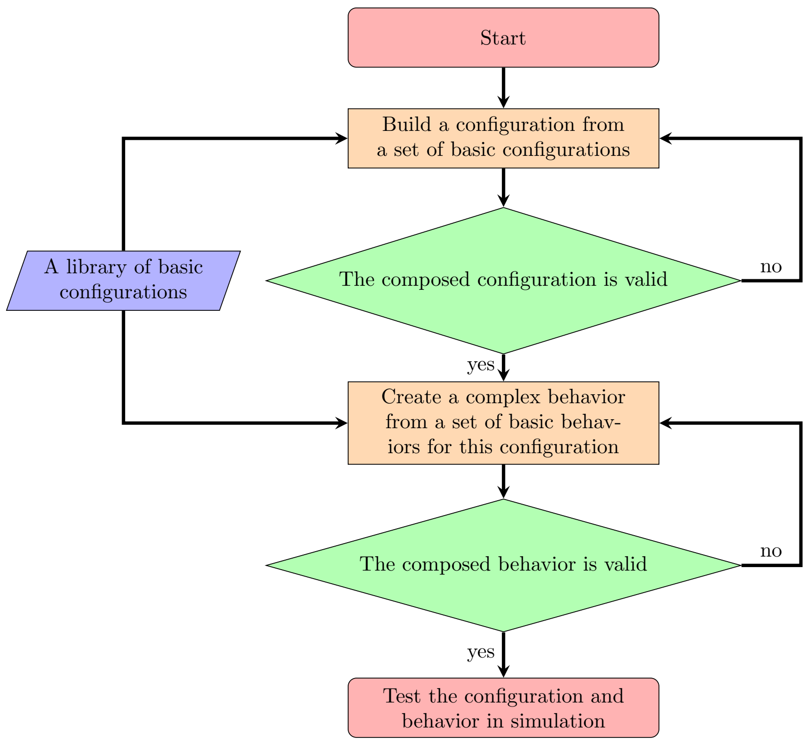
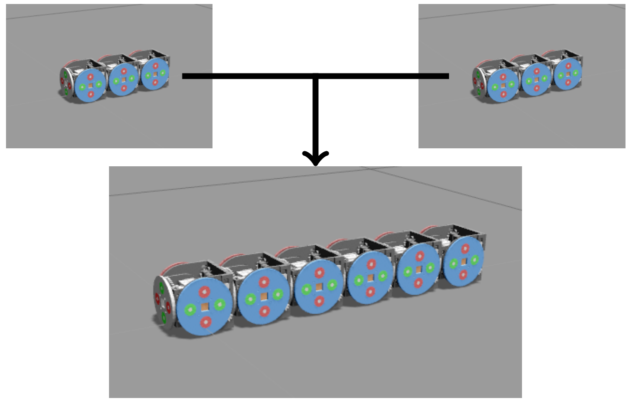


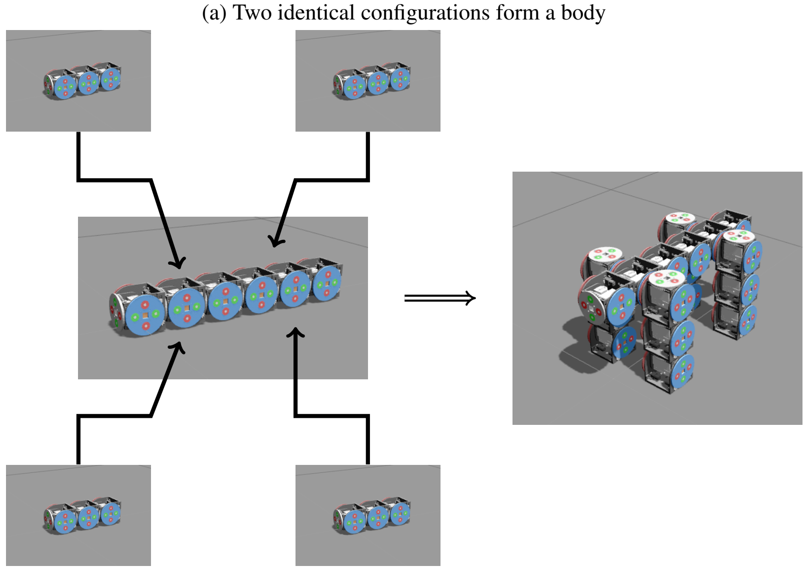
Fig. 7: The design flow

6.3 Scale-up through composition

Our framework allows users to quickly create and program large configurations. The first-order CHAIN3 configuration (Table 1) 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:



(a) Two identical configurations form a body



(b) The body and four identical configurations form a Walkbot

Fig. 8: Building a Walkbot with six identical configurations

$$sineGait = Sc(\ halfSine1,\ halfSine2\ )$$

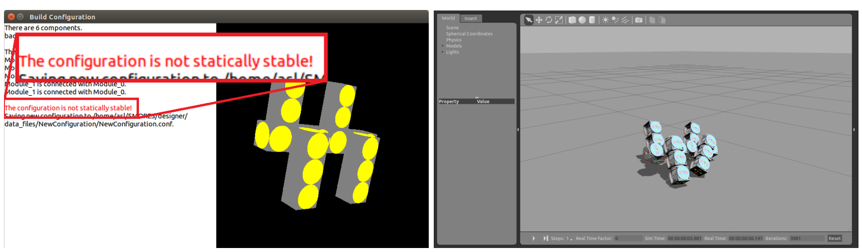
Arbitrarily long snake configurations can be created by composing CHAIN3 configurations end-to-end; Table 2 shows one with 18 modules. A gait for an arbitrarily long snake is created by composing sine wave gaits for each component in parallel, but with alternating phase:

$$nSnakeSineGait =$$
$$Pc\left(Sc\begin{pmatrix} halfSine1 \\ halfSine2 \end{pmatrix}, Sc\begin{pmatrix} halfSine2 \\ halfSine1 \end{pmatrix}, \ldots \right)$$

6.4 Verification

The need for verification becomes more important as design complexity increases. Consider the Walkbot example. If the user sets two of the connections with different angle offset, the composed Walkbot configuration will have two legs pointing in the opposite direction of the other two legs, as shown in Figure 9a. Since the projection of the configuration's center of mass now falls out of the supporting base, the program will warn the user that the configuration is not gravitationally stable. As shown in Figure 9b, in simulation the configuration quickly fell to the ground due to the instability as warned by the program.

Verification of behavior design can also aid the user to create valid and safe robot behaviors. When designing the walking behavior for the Walkbot, if the user commands the front and rear leg at the same side of the robot to swing toward each other at the same time, the program will warn the user that there will be collision between some modules in this behavior, as shown in Figure 10a. The photo shown in Figure 10b demonstrates the moment of collision during simulation.

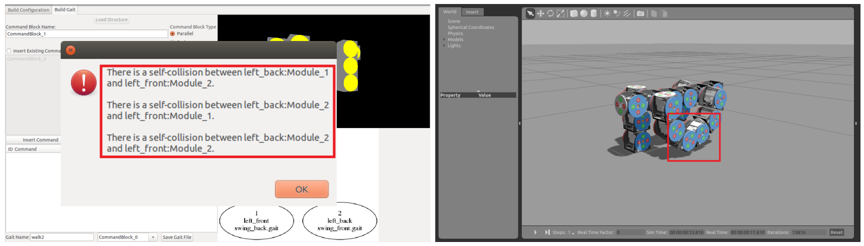


(a) (b)

Fig. 9: (a) The program warns the composed configuration is not gravitationally stable; (b) The robot fell to ground plane due to instability in simulation

7 Results

In the past, designing configurations and behaviors to address new tasks has required time on the order of one to dozens of hours [18]. To evaluate the rapid creation of



(a) (b)

Fig. 10: (a) The program indicates there is collision during the behavior execution; (b) Two feet of the robot collided during simulation

solutions for robotic tasks using modular robot systems, we can look to the modular robot competitions held at large conferences IROS 2003, ICRA 2008, ICRA 2010, ICRA 2011 and ICRA 2012. In each case participants used modular robot systems to solve problems with little a priori knowledge of the task (except that the task would likely require remote operation/manipulation of objects).

In the case of IROS 2003, the task was to create a robot that could “gamble” by placing plastic tokens into a toy slot machine, pulling a lever and catching the chips as they popped out. One team created a solution with PolyBot G1V4 system with 14 modules (14 DoF). Participants had one day to create primitive configurations and behaviors (snake-like mobility, and arm like structures) and 6 hours on the following day to solve the task. No teams completed the task, though some came close. At ICRA 2008, participants created designs to address a mock disaster scenario at a space station on Mars. Teams spent on average about 3 hours building configurations and 3 hours programming them to address the task [18].

Using our framework and library, a single person has created and programmed configurations of similar complexity to those used at the competitions (such as the Walkbot or 18-module snake) in under an hour, or in a few minutes if the right primitives already exist in the library. It would not be fair to compare these times to the competition times directly, since they do not include any hardware tests or hardware debugging. However, we think we can safely say that our framework is an improvement over past design practices.

8 Conclusions

In this paper, we presented a design framework that facilitates the rapid creation of configurations and behaviors for modular robots. Complex configurations are hierarchically constructed from basic subcomponents. We presented a novel motion description language, which allows existing behaviors to be combined in series and parallel to create more complex behaviors. The framework verifies configurations and behaviors as they are being created, allowing early detection of design flaws, specifically self-collision and gravitational instability. After verification, designs can be evaluated in a physical simulator before testing on hardware.

9 Future

Future work will include expansion of the features of our framework. We will add verification tools to check for behavior conflict (when two behaviors composed in parallel command the same DoF), and check for behaviors that would exceed the actuator and connector force limits of the modules. We will also work to include configurations with cycles.

Another area of future work is developing a standard library of configurations and behaviors for the SMORES robot. We will also investigate metrics to evaluate the quality of such a library. Perhaps most importantly, we will test and evaluate the designs and behaviors with actual hardware modules.

Currently, each behavior is associated with exactly one configuration. In many cases, a given behavior could be executed by several different configurations (if it were correctly mapped onto a subset of their modules). In the future, we will apply an embedding detection algorithm (see [12]) to map behaviors into any configuration capable of executing them.

References

[1] Bezzo N, Park J, King A, Gebhard P, Ivanov R, Lee I (2014) Demo abstract: Roslaba modular programming environment for robotic applications. In: Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on, IEEE, pp 214–214
[2] Brockett RW (1988) On the computer control of movement. In: Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on, IEEE, pp 534–540
[3] Casal A (2001) Reconfiguration planning for modular self-reconfigurable robots. PhD thesis, Stanford University
[4] Davey J, et al (2012) Emulating self-reconfigurable robots: Design of the smores system. In: Proc. IEEE Intelligent Robots Systems Conf.
[5] Fukuda T, Kawauchi Y (1990) Cellular robotic system (cebot) as one of the realization of self-organizing intelligent universal manipulator. In: Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on, IEEE, pp 662–667
[6] Hornby GS, Lipson H, Pollack JB (2003) Generative representations for the automated design of modular physical robots. Robotics and Automation, IEEE Transactions on 19(4):703–719

[7] Hristu-Varsakelis D, Anderson S, Zhang F, Sodre P, Krishnaprasad P (2003) A motion description language for hybrid system programming. Institute for Systems Research (preprint)
[8] Huang J, Erdogan C, Zhang Y, Moore B, Luo Q, Sundaresan A, Rosu G (2014) Rosrv: Runtime verification for robots. In: Runtime Verification, Springer, pp 247–254
[9] Kasper Stoy DJC David Brandt (2010) Self-reconfigurable robots: an introduction. MIT Press
[10] Koenig N, Howard A (????) Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, IEEE, vol 3, pp 2149–2154
[11] Lipson H, Pollack JB (2000) Towards continuously reconfigurable self-designing robotics. In: Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on, IEEE, vol 2, pp 1761–1766
[12] Mantzouratos Y, Tosun T, Khanna Y Sanjeev, Mark (2014) On embeddability of modular robot designs. In: IEEE International Conference on Robotics and Automation
[13] Mehta A, Bezzo N, Gebhard P, An B, Kumar V, Lee I, Rus D (2014) A design environment for the rapid specification and fabrication of printable robots. In: nternational Symposium on Experimental Robotics (ISER), Marrakech, Morocco
[14] Melek WW, Goldenberg AA (2003) Neurofuzzy control of modular and reconfigurable robots. Mechatronics, IEEE/ASME Transactions on 8(3):381–389
[15] Pan J, Chitta S, Manocha D (2012) Fcl: A general purpose library for collision and proximity queries. In: Robotics and Automation (ICRA), 2012 IEEE International Conference on, IEEE, pp 3859–3866
[16] Quigley M, Conley K, Gerkey BP, Faust J, Foote T, Leibs J, Wheeler R, Ng AY (2009) Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software
[17] Salemi B, Shen WM, Will P (2001) Hormone-controlled metamorphic robots. In: Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on, IEEE, vol 4, pp 4194–4199
[18] Sastra J (2011) Using reconfigurable modular robots for rapid development of dynamic locomotion experiments
[19] Valdes J, Tarjan RE, Lawler EL (1979) The recognition of series parallel digraphs. In: Proceedings of the eleventh annual ACM symposium on Theory of computing, ACM, pp 1–12
[20] Walter JE, Tsai EM, Amato NM (2002) Choosing good paths for fast distributed reconfiguration of hexagonal metamorphic robots. In: Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on, IEEE, vol 1, pp 102–109
[21] Yim M (1994) Locomotion with a unit-modular reconfigurable robot. PhD thesis, Stanford

[22] Yim M, Duff DG, Roufas K (2000) Modular reconfigurable robots, an approach to urban search and rescue. In: 1st Intl. Workshop on Human-friendly Welfare Robotics Systems, pp 69–76
[23] Yim M, Duff DG, Roufas KD (2000) Polybot: a modular reconfigurable robot. In: Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on, IEEE, vol 1, pp 514–520
[24] Yim M, Shen WM, Salemi B, Rus D, Moll M, Lipson H, Klavins E, Chirikjian GS (2007) Modular self-reconfigurable robot systems [grand challenges of robotics]. Robotics & Automation Magazine, IEEE 14(1):43–52
[25] Yoshida E, Matura S, Kamimura A, Tomita K, Kurokawa H, Kokaji S (2002) A self-reconfigurable modular robot: Reconfiguration planning and experiments. The International Journal of Robotics Research 21(10-11):903–915
[26] Zhang Y, Yim M, Eldershaw C, Duff D, Roufas K (2003) 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, IEEE, vol 3, pp 2442–2447