Computer-aided Compositional Design and Verification 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 automatically identify common problems with behaviors, specifically self-collision and loss of gravitational stability.**

I. Introduction

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

This ~~additional~~ versatility 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 versatile 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 making modularity effective. Significant progress has been made in sharing robotics software between researchers and hardware platforms, most notably ROS[8] which provides IPC and standard libraries for common robot tasks.

The challenges of ~~software design~~ designing controls 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.*~~ *e.g.* 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~~ guide the modularity of our ~~software~~ designs of control. 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 a new ~~scripting language~~ control composition formalism 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 formalism for the construction of modular robot configurations and behaviors by composition, 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 have developed is open-source and will be made freely available online.

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 respects, our work parallels the efforts of Mehta [7] and Bezzo [1], who aim to create and program printable robots from ~~design specification by a novice user~~ 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 ~~MRS~~ 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 (~~i.e.~~ 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 [5]. 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 [13]. The need for new programming techniques to manage the complexity of modular robot systems has been acknowledged in the literature [12]. Historically, gait tables have been a commonly used format in which open-loop kinematic behaviors can be easily encoded [11]. Phased automata have also been presented as a way to easily create scalable gaits for large numbers of modular robots [15]. In this paper, we present a ~~novel scripting language~~ computer aided design that enables novice to quickly create complex behaviors for modular robots.

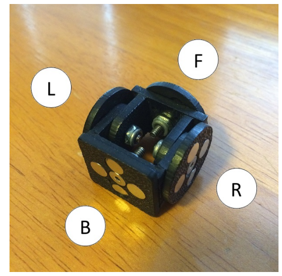
Our framework assists users in verifying design validity by identifying self-collision and loss of gravitational stability. Identification of these conditions is common in modular robot reconfiguration planning [3] and motion planning [14].

IV. Definitions

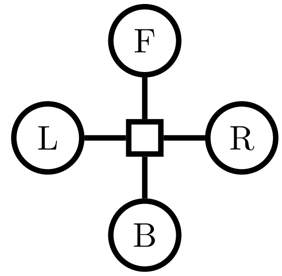
In this section, we present concepts and terms which will be used later in the paper.

Definition IV.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 $\mathcal{M} = ({}^{\mathcal{W}}D^{\mathcal{M}}, X, A, K)$, where:

- ${}^{\mathcal{W}}D^{\mathcal{M}} \in SE(3)$ is the rigid-body *displacement* (position and orientation) of the module body frame in the world reference frame $\mathcal{W}$.
- $X = \{x_1, x_2, \ldots, x_d\}$ 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 module



(b) The graphical representation

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

- $A = \{a_1, a_2, ..., a_k\}$ is the set of *attachment points* where the module can connect to other modules.
- $K : (X, a_i) \to SE(3)$ is the module's *forward kinematics function*, returning ${}^{\mathcal{B}}D^{a_i}$ (the displacement of attachment point a_i in the body frame) as a function of X

Figure 1 shows a module with four DoF and four attachment points

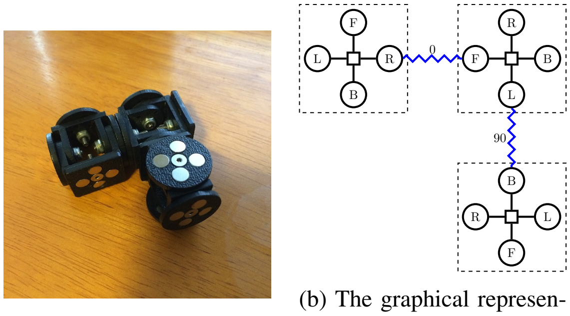
Definition IV.2 (Configuration)**.** A *configuration* is a contiguous set of connected modules which we treat as a single robot. The identity of a configuration is determined by its connective structure; configurations can be represented by graphs with nodes representing modules and edges representing 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 $\mathcal{C} = (C, \gamma, M, E, \delta, X, B)$, where:

- C is a set of sub-configurations, $C = \{\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_q\}$.
- $\gamma : C \to M$ is a function ~~that, when given~~ mapping a configuration~~, returns~~ to all modules of that configuration.
- $M = \bigcup_{\mathcal{C} \in C} \gamma(\mathcal{C})$ is the set of modules.
- E is a set of connections between modules. $(\mathcal{M}_i.a_i, \mathcal{M}_j.a_j) \in E$, where $\mathcal{M}_i, \mathcal{M}_j \in M, \mathcal{M}_i \neq \mathcal{M}_j$, and $a_i \in \mathcal{M}_i.A$, $a_j \in \mathcal{M}_j.A$.
- $\delta : E \to SE(3)$ is a labeling function over connections, returning ${}^{\mathcal{M}_i.a_i}D^{\mathcal{M}_j.a_j}$ the displacement of one attachment point relative the other.
- $X = \bigcup\limits_{\mathcal{M}_i \in M} \mathcal{M}_i.X$ is the *state* of the configuration.
- B is a set of *behaviors* (Definition IV.3) associated with the configuration.

Figure 2a shows a photo of a configuration composed of three ~~SMORES modules~~ modules each with four attachment points. Figure 2b shows its graphical representation. Blue



(a) A configuration

(b) The graphical representation

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

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 majority of functional designs for chain-architecture modular robots are acyclic, so we feel that it does not unreasonably limit the utility of our framework. **TODO: Mark, should we cite something here?**
.

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

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

where $e = (\mathcal{M}_i.a_i,\ \mathcal{M}_f.a_f)$. 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 IV.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 V-B.

Definition IV.4 (Controller)**.** 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.

Definition IV.5 (Behavior Conflict)**.** When writing a behavior, it is possible to command one controller to simultaneously hold more than one desired position; this is known as a *behavior conflict*. Behaviors with conflicts are impossible to execute.

Definition IV.6 (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.

Definition IV.7 (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.

V. Approach and Algorithm

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

A. Configuration Composition

Definition V.1 (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 $\mathcal{C}^*$ that includes all modules and connections from C and E_C.

The set of connections E_C between configurations in C is defined as $(\mathcal{C}_i.\mathcal{M}_i.a_i, \mathcal{C}_j.\mathcal{M}_j.a_j) \in E_C$, where $\mathcal{C}_{i,j} \in C$, $\mathcal{M}_i \in \gamma(\mathcal{C}_i), \mathcal{M}_j \in \gamma(\mathcal{C}_j)$, and $a_i \in \mathcal{M}_i.a$, $a_j \in \mathcal{M}_j.A$. Similar to the assumption about connections in a configuration, we assume that if we form an acyclic graph with configurations in C as nodes and connections in E_C as edges.

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

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

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

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 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 [6]. The atomic commands of our language have unlimited duration (until they are superseded by another command), and each apply to a single DoF of the robot. These properties allow complex behaviors to be created through composition operations.

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~~ 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 **TODO: unfinished**

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 [10]. 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.

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

[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.

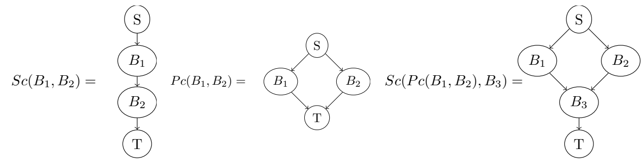


Fig. 3: Series and parallel composition of behaviors

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:

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

Here, θ_0 denotes the currently-sensed value of ~~theta~~ 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:

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

C. *Verification of Configuration and Behavior*

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 4. 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.2 (Verification on Configuration)**.** Given a configuration $\mathcal{C}$ and state X_0, we say $\mathcal{C}$ 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, the mass of each module.

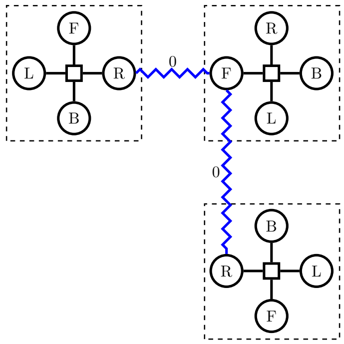


Fig. 4: A configuration with self-collision **TODO: Should we call this self-collision? It's actually topologically flawed, which to me seems different than self-collision.**

a) 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. **TODO: In general, this is a non-trivial problem to solve efficiently. We should briefly mention our method, and cite a reference to other available algorithms.**

b) Stability Checking: Gravitational stability is checked by computing the location of the center of mass of the configuration. For the given configuration $\mathcal{C}$ and state X_0, the position of the center of mass is

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

where $\mathcal{M}_m$ is the mass of the module $\mathcal{M}$, and $P(\mathcal{M})$ is a function returning the position coordinates of $\mathcal{M}$ 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 $\sigma = P(\mathcal{M})_i \forall \mathcal{M}_i \in M_c$ as an approximate set of contact points. If projection of P_c onto the ground plane lies within the convex hull of σ, configuration $\mathcal{C}$ is considered statically stable in state X_0.

2) Verification of Behavior: 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 in two modules colliding during the execution should be considered unsafe.

Definition V.3 (Verification on Behavior)**.** A behavior is valid if it satisfies 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.
- There is no behavior conflict at all time during the execution.
- The configuration is gravitationally stable at the end of the execution.
- The maximum duration when 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 can 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. If there is no behavior conflict, we can update the positions and orientations of all modules in the configuration based on behavior commands. Then we can check collision and gravitational stability of each configuration as we discussed in section V-C1. We argue that this behavior is not safe, if i) there are n consecutive samples when the configuration is not gravitationally stable and $n \cdot t_B > t$; or ii) the configuration at time T_B is not gravitationally stable.

VI. Implementation

We implement a program 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.

A. 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 ~~(DoF)~~ - three continuously rotating faces we call *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 $A = \{L, R, T, B\}$.

B. 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 warn users when the composed configuration is not valid without the usage of a physical simulator, e.g. Gazebo **TODO: Cite Gazebo** .

C. Behavior Builder

Given a composed configuration from the configuration builder, 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

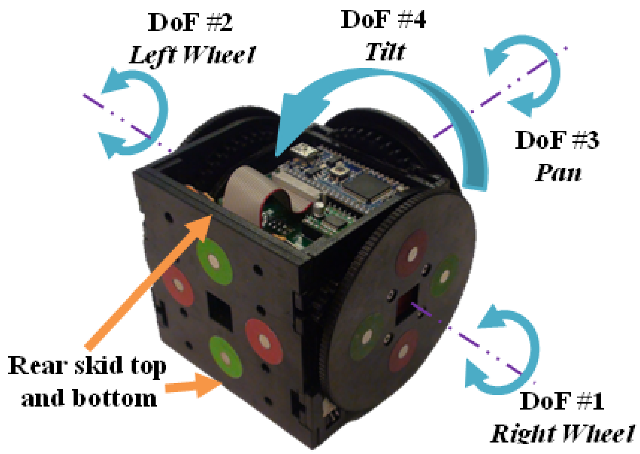
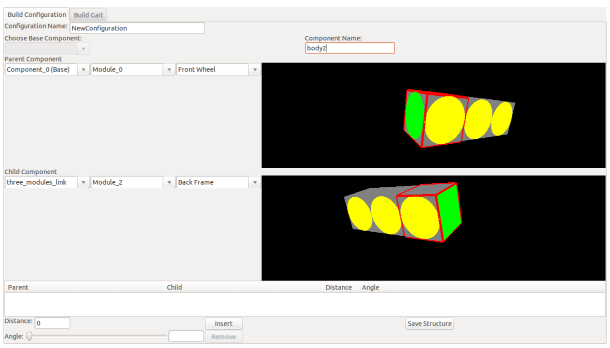
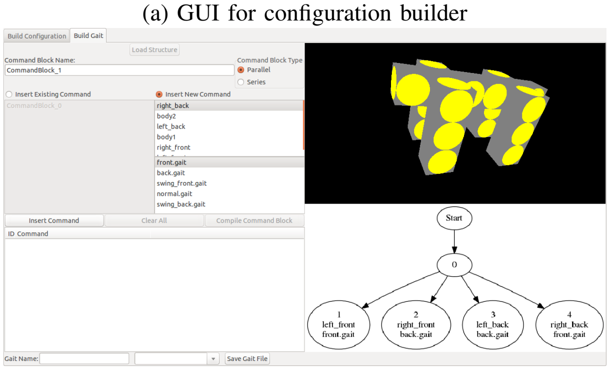


Fig. 5: SMORES robot



(a) GUI for configuration builder



(b) GUI for behavior builder

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

in parallel. Similar to the configuration builder, the behavior builder will also warn users if there is self-collision in the configuration during the execution of composed behaviors without simulations in a physical engine.

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. 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 I and II, it is interested to note that a large and 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.

B. The User Perspective

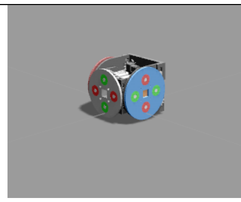
Figure 7 demonstrates the design flow when a user is designing a configuration an its behaviors. We present the start-to-end user perspective in designing a complicated configuration called Walkbot. Consider a basic configuration formed by three modules in a line. We can form a "body" configuration by connecting two of the basic configurations as shown in Figure 8a. 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 8b. 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

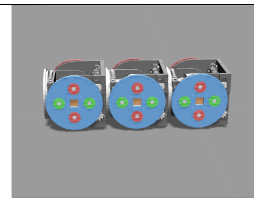
Our framework allows users to quickly create and program large configurations. The first-order CHAIN3 configuration (Figure 9a) can use a *sineGait* behavior to locomote like a snake. Defining *sineGait* as the series composition of two half-waves will allow us to re-use the gait with larger snakes:

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

Arbitrarily long snake configurations can be created by composing CHAIN3's end-to-end; Figure 9b shows one with 18 modules. A gait for an arbitrarily long snake is created by

Config-uration	 DRIVER1
Behaviors	$Drive(v,t)$ $TiltMiddleUp()$

Order-0 (single module)

 CHAIN3
$Drive(v,t)$, $TiltMiddleUp()$, $Step()$, $HoldRigid()$, $Steer(\theta)$, $DriveSideWheels(v,t)$

Order-1

TABLE I: Order-0 and Order-1 configurations

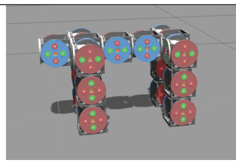
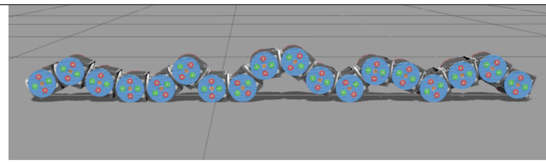
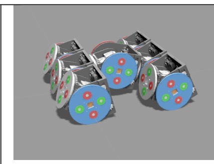
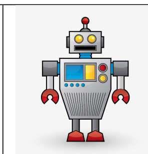
Config-uration	 WALK4	 SNAKE18	 CAR	 DRIVER5
Components	BODY3 LEG3 $\times 4$	SNAKE3 $\times 6$	STEER3 DRIVER1 $\times 4$	STEER3 DRIVER1 $\times 2$
Behaviors	$Walk(t)$	$Slither()$	$DRIVE(v,t)$ $TURN(\theta)$	$DRIVE(v,t)$ $TURN(\theta)$

TABLE II: Order-2 configurations

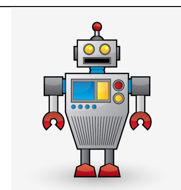
Config-uration	 BACKHOE
Components	PUMA CAR
Behaviors	$Drive(v,t)$ $PickUp()$

TABLE III: Order-3 configurations

composing sine wave gaits for each component in parallel, but with alternating phase:

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

D. Verification

The need for verification becomes more important as design complexity increases. Consider the order-3 Backhoe design, composed of the order-2 car and PUMA arm configurations. It is easy for this design to become gravitationally unstable. Our verification system warns the user when this happens. **TODO: We need to include pictures of this, and flesh out this section.**

VIII. Results

In the past, designing configurations and behaviors to address new tasks has required time on the order of one day [9]. 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
- Mention that we're going to run behaviors on actual hardware.

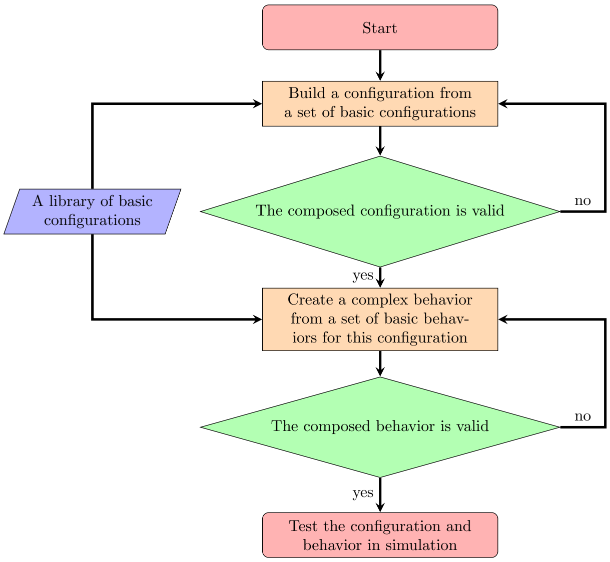
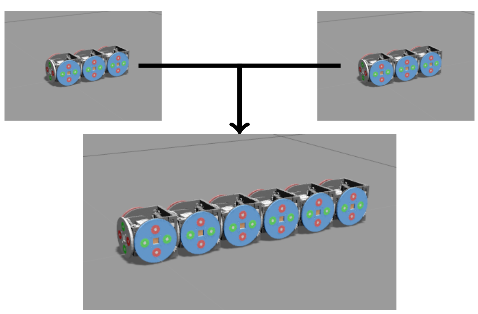
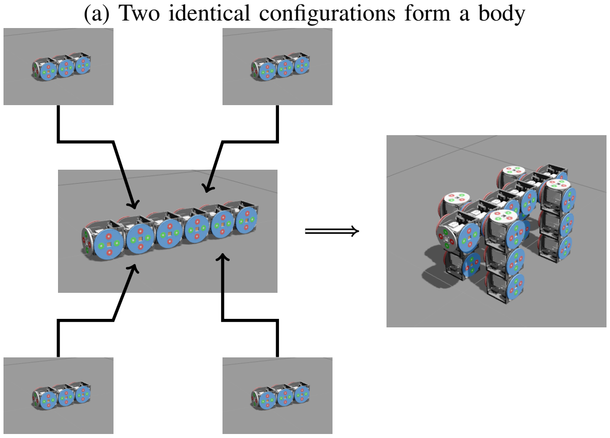


Fig. 7: The design flow

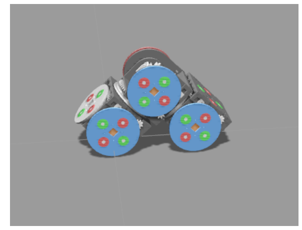


(a) Two identical configurations form a body

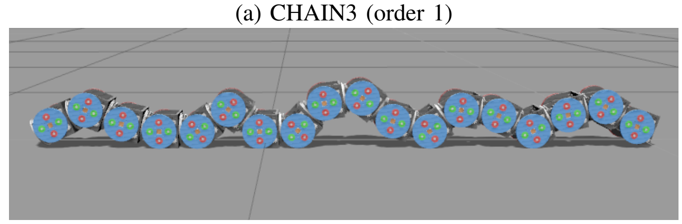


(b) The body and four identical configurations form a Walkbot

Fig. 8: Building a Walkbot with six identical configurations



(a) CHAIN3 (order 1)



(b) SNAKE18 (order 2)

Fig. 9: CHAIN3 and SNAKE18 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] Roger W Brockett. On the computer control of movement. In *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on*, pages 534–540. IEEE, 1988.

[3] Arancha Casal. *Reconfiguration planning for modular self-reconfigurable robots*. PhD thesis, Stanford University, 2001.

[4] J. Davey et al. Emulating self-reconfigurable robots: Design of the smores system. In *Proc. IEEE Intelligent Robots Systems Conf.*, 2012.

[5] 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.

[6] 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.

[7] 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 *nternational Symposium on Experimental Robotics (ISER), Marrakech, Morocco*, 2014.

[8] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[9] Jimmy Sastra. Using reconfigurable modular robots for rapid development of dynamic locomotion experiments. 2011.

[10] 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.

[11] M. Yim. *Locomotion with a unit-modular reconfigurable robot*. PhD thesis, Stanford, 1994.

[12] 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.

[13] 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.

[14] 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.

[15] 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.