Computer-aided Compositional Design and Verification for Modular Robots

Tarik Tosun*, Gangyuan Jing*, Hadas Kress-Gazit, and Mark Yim

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.

Tarik Tosun
University of Pennsylvania, Philadelphia PA, e-mail: tarikt@grasp.upenn.edu

Gangyuan Jing
Cornell University, Ithaca NY, e-mail: gj56@cornell.edu

Hadas Kress-Gazit
Cornell University, Ithaca NY, e-mail: hadaskg@cornell.edu

Mark Yim
University of Pennsylvania, Philadelphia PA, e-mail: yim@grasp.upenn.edu

* Tarik Tosun and Gangyuan Jing contributed equally to this work.

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 [**?**]. Dozens of groups have different kinds of reconfigurable robots[**? ?**], and introduced approaches for programming them [**? ? ?**]. Over 800 papers, a book [**?**], and a survey [**?**] 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 [**?**]. 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 [**?**] which provides inter-process communication and standard libraries for common robot tasks, as well as powerful verification tools [**?**].

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 compo-

nents 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 [**?**] and Bezzo [**?**], 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 [**?**]. Other work has focused on distributed control [**?**] and hormone-based control [**?**].

Efforts have also been made to generate behaviors by automatically identifying the "role" a module should play based on its place in a connected structure [**?**]. Functionality is propagated downward: based on a high-level goal (like "walk") and a connected structure of modules, functional sub-structures (like legs and a spine) are automatically identified, and modules are directed to execute appropriate roles in a distributed fashion. In our work, modular structures are similarly associated with appropriate behaviors. Rather than identifying roles in a top-down fashion, we build designs with the desired functionality from the bottom up. The use creates new designs by composing sub-components and associated behaviors from a library, building a new structure that can definitely execute the desired behavior.

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 [**?**]. The need for new programming techniques to manage the complexity of modular robot systems has been acknowledged in the literature

[**?**]. Historically, gait tables have been a commonly used format in which open-loop kinematic behaviors can be easily encoded [**?**]. Phased automata have also been presented as a way to easily create scalable gaits for large numbers of modular robots [**?**]. In this paper, we introduce a novel motion description language that enables users to quickly create behaviors for modular robots.

A number of robot simulators have been developed, including simulators specifically for modular robots [**?**]. For our work, we opted to use Gazebo [**?**] because of its growing popularity in the robotics community. While our software currently only supports the SMORES robot [**?**], other modular robot designs can easily be incorporated given appropriate SDF files. Future work includes incorporating native support for the ckBot robot [**?**].

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 [**?**] and motion planning [**?**]. 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 $\mathscr{M} = ({}^{\mathscr{W}}D^{\mathscr{M}}, X, A, K)$. The rigid body *displacement*, ${}^{\mathscr{W}}D^{\mathscr{M}} \in SE(3)$ gives the position and orientation of the module body frame in the world frame $\mathscr{W}$. The *state* of the module, $X = [x_1, x_2, \ldots, x_d]$, is a d-dimensional vector representing 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. The module's *forward kinematics function*, $K : (X, a_i) \rightarrow SE(3)$ returns ${}^{\mathscr{B}}D^{a_i}$ (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.

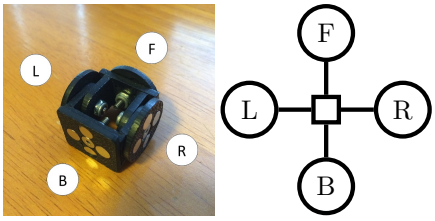


Fig. 1 A photo of a module (left) and its graphical representation (right)

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 deter-

mined 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 $\mathscr{C} = (C, \gamma, M, E, \delta, X, B)$. Here, $C = \{\mathscr{C}_1, \mathscr{C}_2, ..., \mathscr{C}_q\}$ is a set of sub-configurations. $\gamma : C \to 2^M$ is a function mapping a configuration $\mathscr{C}_i \in C$ to its set of modules, $M = \bigcup_{\mathscr{C} \in C} \gamma(\mathscr{C})$. E is a set of connections between modules. Elements of E are pairs of attachment points, $(\mathscr{M}_i.a_i, \mathscr{M}_j.a_j) \in E$, where $\mathscr{M}_i, \mathscr{M}_j \in M, \mathscr{M}_i \neq \mathscr{M}_j$, and $a_i \in \mathscr{M}_i.A$, $a_j \in \mathscr{M}_j.A$. The orientation of one attachment point relative another is represented by the labeling function $\delta : E \to SO(3)$, returning. The *state* of the configuration is $X = \bigcup_{\mathscr{M}_i \in M} \mathscr{M}_i.X$. Finally, associated with each configuration is a set of *behaviors* B (see Definition 3).

Figure 7 shows a photo of a configuration composed of three modules, each with four attachment points, and its graphical representation. Blue zigzag lines represent connections between modules, and the label of each connection shows the angle offset of that connection. 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.

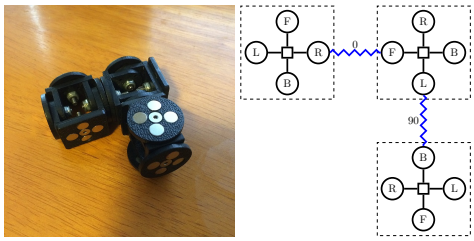


Fig. 2 A photo of a configuration with three modules (left) and its graphical representation (right).

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.

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.

Definition 7 (Maintain Connections and Joint Angles). Due to the force limit of mechanisms in each module, there maybe situations, especially during execution of a behavior, when force provided is not sufficient to maintain all connections between modules in a configuration, or to maintain all joints at desired positions.

4 Approach and Algorithm

The three major components of our framework are configuration composition, behavior composition, and verification of configurations and behaviors. Together, these three components provide a streamlined workflow to quickly create functional robots by leveraging an existing library of designs and behaviors. Combining existing designs and behaviors into new ones allows users to create large, complicated, and highly functional designs.

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 8 (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. we define the composed configuration to be $\mathscr{C}^* = (C^*, \gamma, M, E, \delta, X, B)$, where C^*, M, E, and B are defined as the unions of the corresponding sets of the sub-configurations in C. The definitions of γ, X, and δ remain unchanged from Definition 2.

4.2 Behavior Composition: Series-Parallel Action Graphs

The modular robotics community has developed a number of languages and methods to create behaviors, including gait tables [**?**], phased automata [**?**], hormone-based control [**?**], and role-based control [**?**]. Phased automata, hormone, and role-based control are typically used to specify a single, cyclic behavior (such as a gait for locomotion) in a distributed fashion. These methods have good scaling and robustness properties, but are not well-suited to specifying the non-cyclic, globally coordinated behaviors required for many tasks (like picking up and moving an object with an arm).

The simplicity and clarity of gait tables makes them appealing for our application. However, gait tables are difficult to re-use, and also hard to scale to very complicated designs. The motion description language we present allows low-level behaviors to be combined in series and parallel to create new higher-level behaviors, encapsulating complexity and facilitating code re-use. The resulting programs are precise and expressive, and have a nested structure that is easy to understand and debug.

The atoms of the language are called actions. Similar to a single entry of a gait table, an action specifies a controller setpoint for a single DoF of a module. Unlike a gait table entry, actions do not have explicit timestamps. Rather, each action has an associated interrupt condition, which is a boolean function of the (sensed) state of the robot. Similar to a state transition in a finite state machine (FSM), when the interrupt condition is met the action is considered complete, and execution moves on to the next actions. Interrupts allow the programmer to precisely specify behaviors in a natural way: rather than specifying a timed sequence of motions, the programmer specifies an ordered sequence of actions and has some assurance that an action will not begin until the robot has actually achieved the goal state of the previous action. Actions may optionally include a timeout, which causes the action to be considered complete automatically once time runs out.

An important distinction between actions in our language and states in a traditional FSM is that multiple actions may execute in parallel. Actions are combined through parallel and series composition to create behaviors. When two actions are composed in series, the second begins when the first ends. When composed in parallel they begin simultaneously, and the following actions do not begin until both complete. A behavior created using these operations is a directed acyclic graph of actions with series-parallel structure [**?**]; Figure 3 provides a visual example. This structure is advantageous because the series and parallel composition operations

remain valid for behaviors of arbitrary size, allowing new behaviors to be easily created by combining existing behaviors.

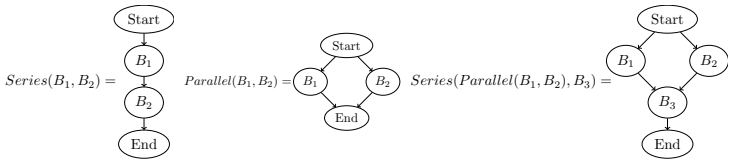


Fig. 3: Series and parallel composition

As an example, consider the car design shown in figure 4. To create a low-level "drive-forward" behavior, we simply command all of its wheels to spin in parallel. The car steers by swiveling its central steering column, so a "turn right" behavior can be similarly achieved by commanding parallel actions for the steering column joints. With these low-level behaviors established, we can command trajectories through series composition, for example: *square=series(drive, turn, drive, turn, drive, turn, drive, turn)*.

This paradigm is lightweight enough to code low-level behaviors quickly and easily. However, the real value comes from its ability to combine behaviors in layers and quickly generate behaviors for complicated designs. This works particularly well when designs are made by composing smaller designs. For example, we can easily develop "drive" and "turn" behaviors for the 18-module backhoe shown in Figure 4 by composing behaviors of its component car designs: $driveBackhoe = parallel(driveCar1, driveCar2)$, and $turnBackhoe = parallel(turnCar1, turnCar2)$. Each of these one-line statements commands the movement of 28 degrees of freedom.

We can extend this further to generate a useful high-level behaviors for the backhoe. Suppose that the arm has a laser rangefinder attached to the end, and that we've already created a "scan" behavior that sweeps or rotates the sensor. We might create a "patrol" behavior that scans continuously while driving in a square: $patrol = parallel(scan, square)$. Or, if we only want the robot to scan the corners of a room, we can precisely specify this using lower-level behaviors: $cornerScan = series(drive, scan, turn, drive, scan, turn, \ldots)$.

Building behaviors in a layered fashion make it much easier to re-use code. If primitive behaviors have not yet been built, building a single complicated behavior like "patrol" from scratch might be faster than first building primitive behaviors and then composing them into the desired complex behavior. However, the latter process is more efficient in the long run, because the low-level behaviors can be re-used to quickly create many different high-level behaviors.

Our emphasis on abstraction begs the question: why not use a a more fully-featured plan execution model such as behavior trees [**?**], or even a traditional object-oriented programming language (like Java or C++)? Our decision was driven by the tradeoff between complexity and ease-of-use: given our desire for a simplic-

ity and speed of programming, we chose to use an extremely minimal paradigm with only two composition operations. The language is quite expressive: we have used it to develop a number of complex behaviors for large designs (see Section 6). In the future, we hope to expand its capabilities without sacrificing ease-of-use. In particular, facilities for high-level decision making based on sensed information about the environment will be a priority.

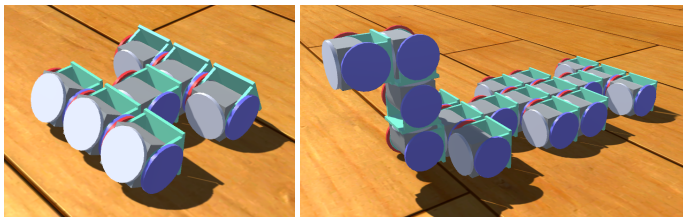


Fig. 4: Car design (left) and backhoe design (right)

4.3 Verification of Configurations and Behaviors

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 modules. Consider the configuration shown in Figure 5. 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.

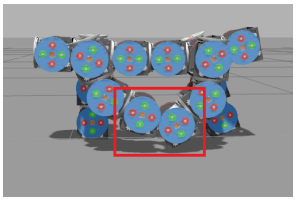


Fig. 5 A configuration with self-collision **TODO: We might consider taking this figure out to save space. I think self-collision is easy enough to understand without a picture**

Definition 9 (Configuration Verification). Given a configuration $\mathscr{C}$ and state X_0, we say $\mathscr{C}$ is valid in state X_0 if it is gravitationally stable and free from self collision between modules.

We will now describe the methods used to identify gravitational instability and self-collision. We have implemented these methods for the SMORES robot, but could easily apply them to other robots given appropriate geometric and mass information for each module. To check for self-collision, the positions and orientation of all modules are obtained through forward kinematics. 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 [**?**]. Gravitational stability is checked by computing the location of the center of mass of the configuration based on the known kinematics and mass properties of the modules. We find the set of modules that have minimal position in the z direction and consider them to be in contact with the ground plane, treating their centroids as an approximate set of ground contact points. If the projection of the configuration center of mass onto the ground plane lies within the convex hull of the ground contact points, the configuration is considered statically stable in state X_0.

4.3.2 Verification of Behaviors

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 10 (Behavior Verification). A behavior is valid if: there are no collisions or behavior conflicts at any time during execution, and forces required to maintain all module connections and joint positions do not exceed their limit. Additionally, the maximum duration for which the configuration is not gravitationally stable during execution is less than a time bound t_{max}, and the configuration must be stable at the end of execution.

For a behavior with time duration 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.

For checking force limit, we create the model for a module and specify its joint and connection mechanisms in PhysX [**?**] physics engine with a graphical designing tool called Unity3D [**?**]. By setting the maximum force allowed for the connection between modules and for controlling each joint in the module, we are able to identify a unsafe behavior if we detect, during the behavior execution, any undesired module

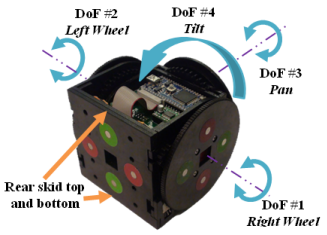


Fig. 6 SMORES robot

disconnection or any joint's mismatch between its target position and steady state position.

5 Implementation

A design interface was implemented to aid users in building complex configurations and behaviors from a set of basic configurations and associated behaviors, and then verifying their correct execution. 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 any other modular robot for which an SDF description is available.

5.1 SMORES robot

We demonstrate our system using the SMORES modular robot, developed at the University of Pennsylvania [**?**]. Each SMORES modules has four DoF - three continuously rotating faces called *turntables* and one central hinge with a 180° range of motion (Figure 6). 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 in Figure 7. In addition, the configura-

tion builder will warn users when the composed configuration is not valid without the usage of a physical simulator, e.g. Gazebo [**?**].

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 7 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.

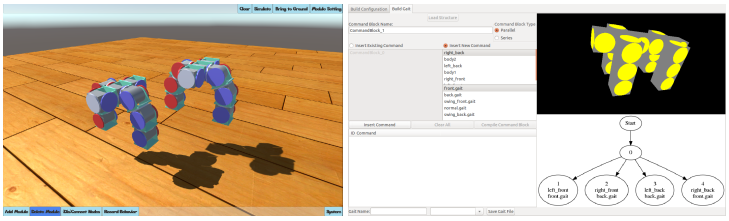


Fig. 7: GUIs for configuration builder (left) and behavior builder (right)

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.

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.

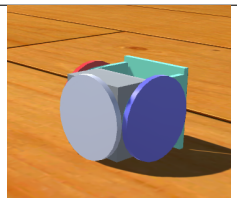
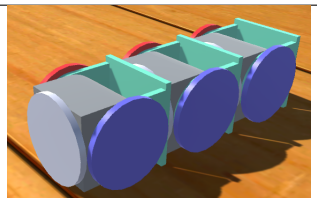
	Order-0 (single module)	**Order-1**
Configuration	 MODULE	 CHAIN3
Behaviors	$Drive(v,t)$, $TiltMiddle(\theta)$, $SpinTop(\theta)$	$Drive(v,t)$, $SteeringPose()$, $LegStep()$, $HoldRigid()$, $Steer(\theta)$

Table 1: Order-0 and Order-1 configurations

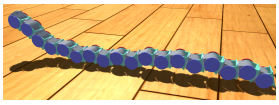
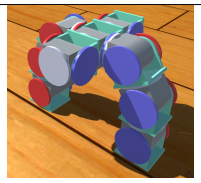
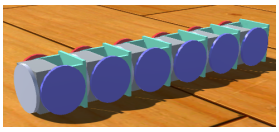
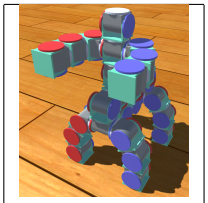
	Order-2			**Order-3**
Configuration	 CHAIN18	 CROSS	 CHAIN6	 CENTAUR
Components	CHAIN3 $\times 6$	CHAIN3 $\times 3$	CHAIN3 $\times 2$	CROSS $\times 3$ MODULE $\times 2$
Behaviors	$SineGait18()$		$Drive(v,t)$, $Turn(\theta)$ $HoldRigid()$	$Walk()$

Table 2: Order-2 and Order-3 configurations

6.2 The User Perspective

Figure 8 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 Centaur. Consider a basic configuration "CHAIN3" as shown in Table 1. We can form a "CROSS" configuration. With two "CROSS" configuration, we can form a body, as demonstrated in Figure 9a. With three more configurations attaching to the top of the body configuration with certain angle offset, we can build a complex configuration, Centaur, with four legs, as shown in Figure 9b. Notice that by connecting multiple basic configurations together, we can design a complex configuration more easily than creating with individual modules.

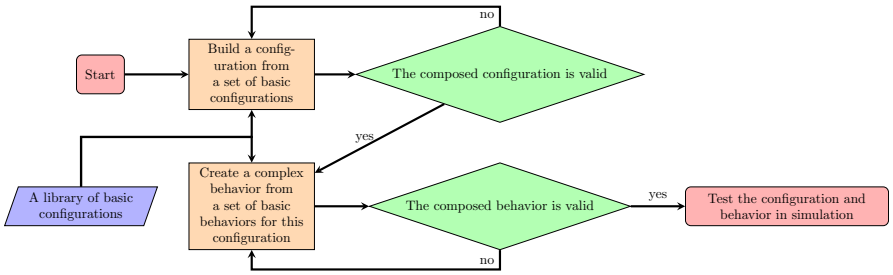
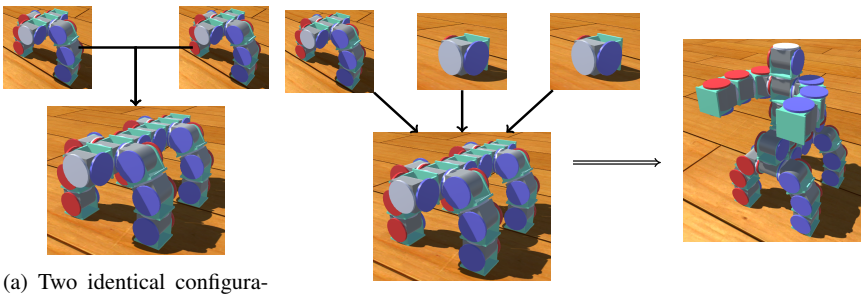


Fig. 8: The design flow



(a) Two identical configurations form a body

(b) The body and other configurations form a Centaur

Fig. 9: Building a Centaur with a set of configurations

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:

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

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:

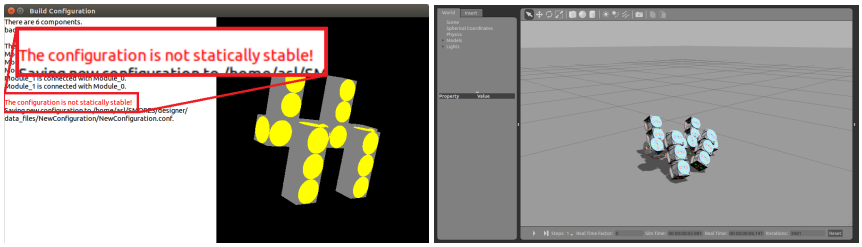
$$nSnakeSineGait = Pc\left(\ Sc\begin{pmatrix} halfSine1 \\ halfSine2 \end{pmatrix},\ Sc\begin{pmatrix} halfSine2 \\ halfSine1 \end{pmatrix}, \dots\ \right)$$

6.4 Verification

The need for verification becomes more important as design complexity increases. Consider a four-legged 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 10a. 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 10b, in simulation the configuration quickly fell to the ground due to the instability as warned by the program.

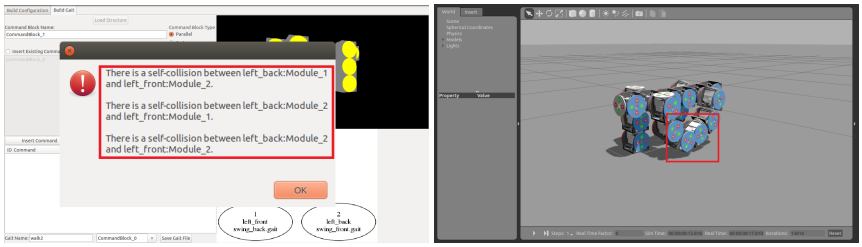
Verification of behavior design can also aid the user to create valid and safe robot behaviors. When designing the walking behavior for the Walkbot, if the user commands the front and rear leg at the same side of the robot to swing toward each other at the same time, the program will warn the user that there will be collision between some modules in this behavior, as shown in Figure 11a. The photo shown in Figure 11b demonstrates the moment of collision during simulation.

Despite the design of the modular robot system, modules all have limits on maximum force that is available to maintain connections with other modules and to drive each joint to desired positions. Thus, it is crucial to notify the user if there is no sufficient force from module's hardware to execute a behavior while maintaining all modules connections. As shown in Figure 12a, the program detected an undesired disconnection when the user tried to lift a long cantilever arm. Figure 12a demonstrates the disconnection in simulation.



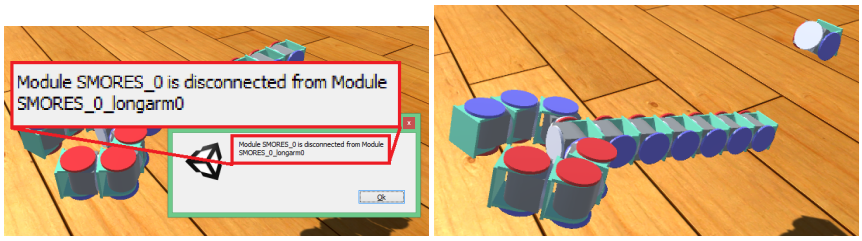
(a) (b)

Fig. 10: (a) The program warns the composed configuration is not gravitationally stable; (b) The robot fell to ground plane due to instability in simulation



(a) (b)

Fig. 11: (a) The program indicates there is collision during the behavior execution; (b) Two feet of the robot collided during simulation



(a) (b)

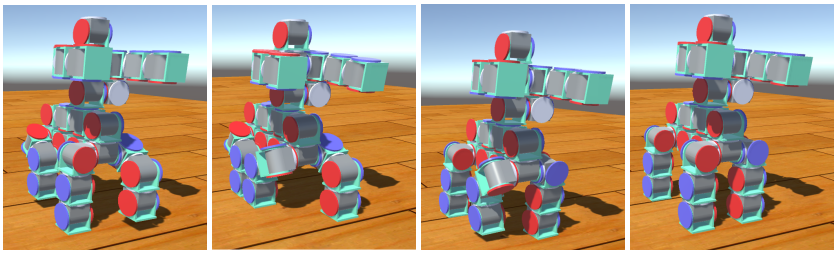
Fig. 12: (a) The program indicates there is undesired disconnection; (b) The configuration disconnected during simulation

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 [**?**]. To evaluate the rapid creation of 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 [**?**].

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 Centaur or 18-module snake) in under an hour, or in a few minutes if the right primitives already exist in the library. Figure 13 shows a four-step walking behavior for the Centaur. 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.



(a) (b) (c) (d)

Fig. 13: A four-step walking behavior

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 [**?**]) to map behaviors into any configuration capable of executing them.