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 and verify 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. In addition to verification, users can evaluate their configurations and behaviors in a physics-based simulator.

1 Introduction

Modular reconfigurable robot systems have been studied extensively for several decades. These systems distinguish themselves from conventional robotic systems

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.
This work was funded by NSF grant number CNS-1329620.

in their ability to transform into different shapes to address a wide variety of tasks. They promise to be versatile, robust, and low cost [27]. Dozens of groups have built different kinds of reconfigurable robots [8, 13], and introduced approaches for programming them [19, 22, 30]. Over 800 papers, a book [11], and a survey [28] have been written on the subject.

This versatility places an additional burden on the user, because solving problems with modular robots involves not only writing programs, but also determining 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 [26]. 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. Significant progress has been made on these fronts in traditional robotics, most notably ROS [18] which provides inter-process communication and standard libraries for common robot tasks, as well as verification tools [10]. In modular robotics, the challenge is different. Modular robot systems are not usually optimized for specific tasks, so in order to use them most effectively, we must take 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 them while they are being created. New configurations are created by combining existing sub-configurations, for example combining a four-legged walking robot with a two-fingered gripper to form a mobile manipulator, like the "Centaur" configuration shown in Figure 1 . Behaviors are associated with each configuration, so that when sub-configurations are composed, their associated behaviors are immediately available for use. The Centaur in Figure 1, for example, can immediately execute the walking behavior of its component four-legged base. We introduce a new motion description language (Series-Parallel Action Graphs, Section 4.2) that facilitates the rapid creation of complex behaviors by composition of simpler behaviors (for example, composing "Grasp" and "Walk" behaviors to make the Centaur pick up and carry an object). We provide tools that automatically verify configurations and behaviors during the design process, identifying conflicting commands, self-collision, loss of gravitational stability, and forces exceeding the limits of safety for actuators and connectors. This allows users to identify problems early and iterate quickly on complex new designs. In addition to verification, users can evaluate their configurations and behaviors in a physics-based simulator. The software we have developed is open-source, and will be made freely available online at: `http://modlabupenn.org/compositional-design/`.

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.

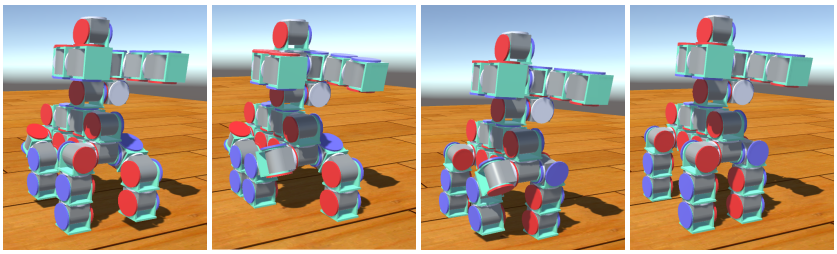

Fig. 1: The Centaur is a mobile manipulator made of 29 modules. The framework we present provides tools that help users quickly create, program, and verify complex designs like the Centaur by composing existing designs and behaviors from a library.

2 Related Work

In some respects, our work parallels the efforts of Mehta [16] and Bezzo [3], 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 configuration for every new task - configurations 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. Genetic algorithms have been applied for the automated generation of designs and behaviors [9]. Other work has focused on distributed control [24], hormone-based control [19], and central pattern generators [21].

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 [22]. 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 user 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 [28]. The need for new programming techniques to manage the complexity of modular robot systems has been acknowledged in the literature [26]. Historically, gait tables have been a commonly used format in which open-loop kinematic behaviors can be easily encoded [25]. Phased automata have also been presented as a way to easily create scalable gaits for large numbers of modular robots [30]. 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 [5]. For our work, we opted to use Gazebo [12] because of its growing popularity in the robotics community. While our software currently only supports the SMORES robot [7], other modular robot designs can easily be incorporated. Future work includes incorporating support for the CKBot robot [6].

Our framework assists users in verifying design validity by identifying conflicting commands, self-collision, loss of gravitational stability, and forces exceeding the limits of safety for actuators and connectors. In existing literature, some of these conditions have been checked in the context of modular robot reconfiguration planning [4] and motion planning [29]. 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 joint angles 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 ${}^{\mathscr{M}}D^{a_i}$ (the displacement of attachment point a_i in the

module frame) as a function of X. Figure 2 shows a schematic representation of a module with four attachment points.

In this paper, we demonstrate our framework using a homogeneous modular robot system (all modules are identical). The framework could be extended to heterogeneous systems by including more information in the definition of a module - for example, if the system used multiple kinds of connectors, labels on the attachment points could be included.

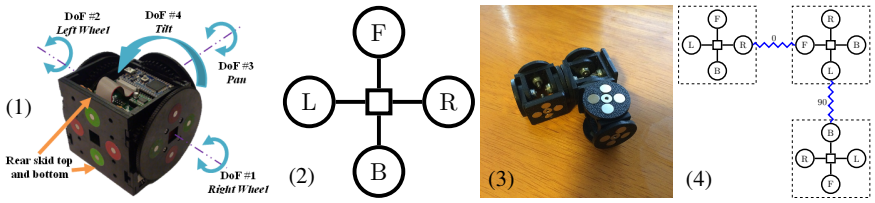

Fig. 2: From left: (1) A photo of a SMORES module with four attachment points (left, right, front, and back), (2) its graphical representation, (3) a photo of a configuration with three modules, and (4) its graphical representation.

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 type).

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 to another is represented by the labeling function $\delta : E \to SO(3)$, returning ${}^{\mathscr{M}_i.a_i}R^{\mathscr{M}_j.a_j}$. 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 2 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 ${}^{\mathscr{W}}D^{\mathscr{M}_f}$ 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* $B : (t,X) \rightarrow X_{set}$ is a programmed sequence of movements defined over the joints of a specific configuration, and intended to produce a desired effect - a gait for walking is one example. Behaviors determine the controller setpoints X_{set} for a configuration as a function of state X and time t. In this paper, we represent behaviors 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.

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, functional designs.

4.1 Configuration Composition

Before discussing 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 $a_i \in \mathscr{M}_i.A$, $a_j \in \mathscr{M}_j.A$. We form a graph with configurations in C as nodes and connections in E_C as edges.

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. The composed configuration is $\mathscr{C}^* = (C^*, \gamma, M, E, \delta, X, B)$, where C^*, M, E, and B are the unions of the corresponding sets of the sub-configurations in C.

4.2 Behavior Composition: Series-Parallel Action Graphs

The modular robotics community has developed a number of methods to create behaviors, including gait tables [25], phased automata [30], hormone-based control [19], and role-based control [22]. 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 often difficult to compose or 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 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 action. 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 [23]; Figure 3 provides a visual example.

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

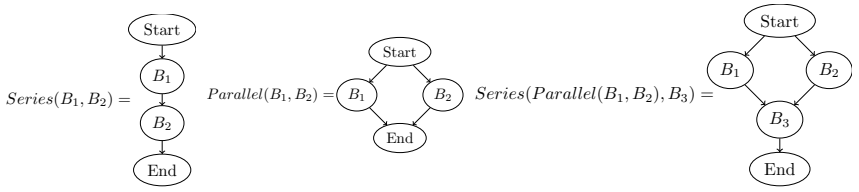

Fig. 3: Series and parallel composition

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, if we name our car configuration `c`: `c.square = series(c.drive, c.turn, c.drive, c.turn, c.drive, c.turn, c.drive, c.turn)`.

This paradigm allows low-level behaviors to be coded 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 develop "drive" and "turn" behaviors for the 18-module backhoe shown in Figure 4 (name `bh`) by composing behaviors of its component car designs (named `c1` and `c2`): `bh.drive = parallel(c1.drive, c2.drive)`, and `bh.turn = parallel(c1.turn, c2.turn)`. Each of these one-line statements commands the movement of 28 degrees of freedom.

We can extend this further to generate 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: `bh.patrol = parallel(bh.scan, bh.square)`. Or, if we only want the robot to scan the corners of a room, we can precisely specify this using lower-level behaviors: `bh.cornerScan = series(bh.drive, bh.scan, bh.turn, bh.drive, bh.scan, bh.turn, ...)`.

Building behaviors in this layered fashion makes it easy to re-use code and quickly generate complicated behaviors. Of course, there is no guarantee that two composed behaviors will be compatible; it is possible to mistakenly create behaviors that are impossible or dangerous to execute. For this reason, we provide verification tools that automatically identify problems - for example, if two behaviors composed in parallel commanded the same DoF simultaneously (a problem we call *behavior conflict*), this would be automatically identified. Our verification tools are explained in detail in Section 4.3.

Our emphasis on abstraction begs the question: why not use a a more fully-featured plan execution model such as behavior trees [15], parallel-hierarchical finite state machines [20] 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 simplicity and speed of programming, we chose a

minimal paradigm with only two composition operations. The language is quite expressive: we have used it to develop complex behaviors for large designs (see Section 6). The language is also limited: it does not yet include conditional statements, iteration, or access to environmental sensor information (other than joint angles). In the future, we hope to include these capabilities without sacrificing ease-of-use.

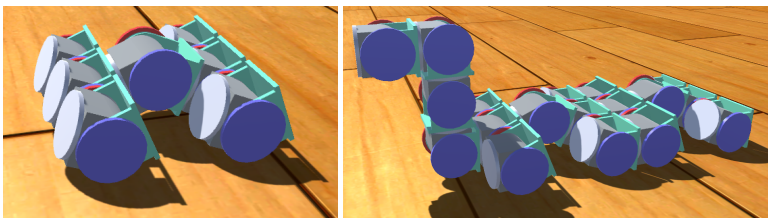

Fig. 4: Car design (left) and backhoe design (right)

4.3 Verification of Configurations and Behaviors

Verification of Configurations: In Section 4.1, we introduced the definition of configuration composition. During the design process, a user might attempt to compose configurations in a way that is unstable or physically impossible. By incorporating existing algorithms into the design process, we provide tools to automatically verify designs during construction, saving time that would otherwise be spent simulating or testing invalid designs.

Given a configuration $\mathscr{C}$ and state X_0, we consider $\mathscr{C}$ to be valid in state X_0 if it is gravitationally stable and free from self collision between modules. A robot is gravitationally stable when it is balanced, and gravity does not create any net moment on it. If this condition is not met, the robot could tip over and suffer damage. A *self-collision* occurs when two different parts of the configuration are commanded to occupy the same location in space. Self-collisions can also cause damage, and are almost always unwanted.

To determine gravitational instability and self-collision, we assume that the geometric, kinematic, and mass information for each module are available. To check for self-collision, the positions and orientation of all modules are obtained through forward kinematics as in Definition 2. Our tool checks self-collision by approximating modules as spheres, and checking the distance (radius) between all pairs. Due to this approximation, false-positive collisions might be detected. When this happens, a user can easily spot the faulty detection in the final configuration and choose to ignore such warning. More sophisticated techniques are available which efficiently produce exact results [17], at the cost of higher complexity. To offer instant feed-

back to the user when designing the configurations, we check gravitational stability 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, gravity exerts no moment and the configuration is stable.

Verification of Behaviors: In Section 4.2, we introduced a novel motion description language for modular robots. Like configurations, behaviors are automatically verified as the user composes them. In addition to being free from self-collision and gravitationally stable during execution, a valid behavior also must not exceed the actuator or connector force limits of the modules, and must be free from behavior conflict.

To verify a behavior with time duration T_B, we discretize execution with a preset sampling time t_B. At each time step, 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 the commands. We then check for self-collision and gravitational instability, using the methods described above. A behavior that results in self-collision during a single time step is considered invalid. For gravitational stability, we specify a time bound $t_{max} > t_B$. A behavior is considered unstable if it includes any period of instability longer than t_{max}, or if the behavior is unstable at time T_B (at the end).

To check force limits, unlike other verifications for behaviors, we use an existing physics engine to detect unsafe conditions during simulation. By setting the maximum force that can be supported by connectors and exerted by joints, we are able to identify unsafe behaviors if we detect, during the behavior execution, any undesired module disconnection or a mismatch between any joints target position and actual position.

The need for verification becomes more important as design complexity increases. Consider a four-legged Walkbot example shown in Figure 5-1. 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 5-1. 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 5-2, in simulation the configuration quickly fell to the ground due to the instability as warned by the program.

Verification of behaviors also aids the user in creating 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 modules in this behavior, as shown in Figure 5-3. The image shown in Figure 5-4 demonstrates the moment of collision during simulation.

There is a trade-off between the correctness and the efficiency of the verification. By reducing the sampling time t_b, more potential self-collisions or gravitational instability can be detected with the cost of longer computation time. However, since there is no real-time requirement (verification is done during the design process, not at runtime), the computational cost of fine-resolution verification is worthwhile in most cases.

Modules all have limits on the 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 the module's hardware to execute a behavior while maintaining all module connections. As shown in Figure 5-5, the program detected a disconnection when the user tried to lift a long cantilever arm. Figure 5-6 demonstrates the disconnection in simulation.

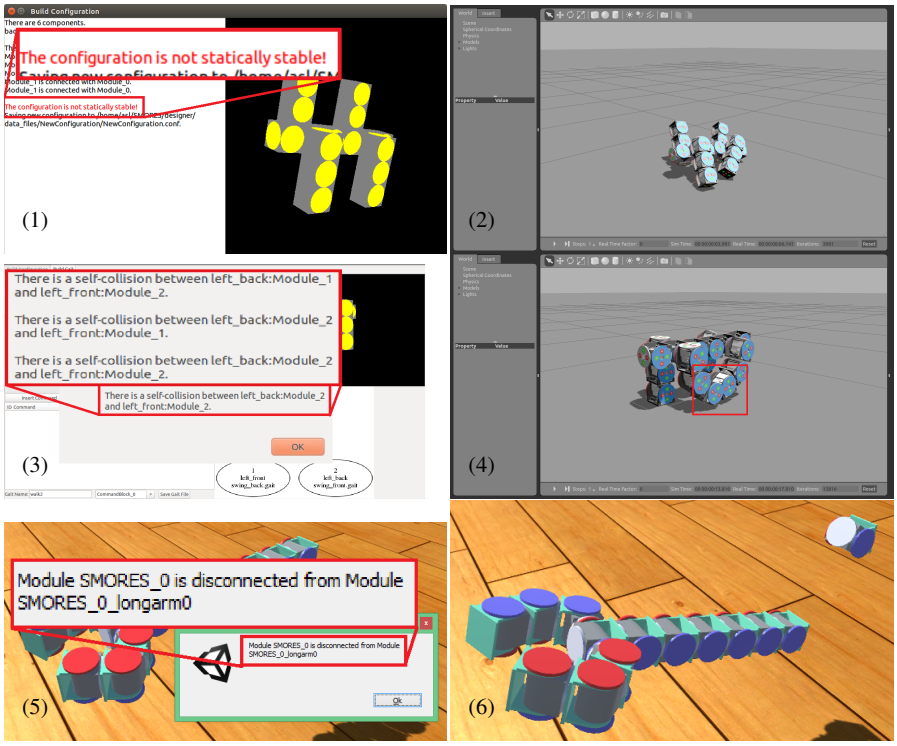

Fig. 5: From top and left: (1) The design tool warns the composed configuration is not gravitationally stable; (2) The robot fell to ground plane due to instability, in simulation; (3) The design tool indicates there is collision during the behavior execution; (4) Two feet of the robot collided during simulation; (5) The design tool indicates there is undesired disconnection; (6) The configuration disconnected during simulation.

5 Implementation

Our implementation currently supports only the SMORES modular robot [7], but could easily incorporate any other modular robot for which kinematic, geometric, and mass information is available. Each SMORES modules has four DoF - three continuously rotating faces called *turntables* and one central hinge with a 180° range of motion (Figure 2-1). The DoF marked 1, 2, and 4 have rotational axes that are parallel and coincident. Each SMORES module can drive around as a two-wheel differential drive robot. SMORES modules may connect to one another via magnets on each of their four faces, and are capable of self-reconfiguration. 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 $A = \{L, R, T, B\}$.

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 design tool into two main parts: a configuration builder and a behavior 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 nodes on each configuration, as demonstrated in Figure 6. In addition, the configuration builder warns users when the composed configuration is not stable or contains self-collisions. It does so without the computation complexity of a physical simulator, e.g. Gazebo [12].

Given a configuration, the behavior builder aids users in designing behaviors by composing existing behaviors in series and parallel. Figure 6 illustrates a new behavior composed by putting four basic behaviors in parallel. Similar to the configuration builder, the behavior builder warns users if there are self-collisions or behavior conflicts during the execution of composed behaviors, without the need of a physics-based simulator. To check force limits for connections and actuators, we create a model of the module in the PhysX [1] physics engine with Unity3D [2], and specify joint and connection force limits.

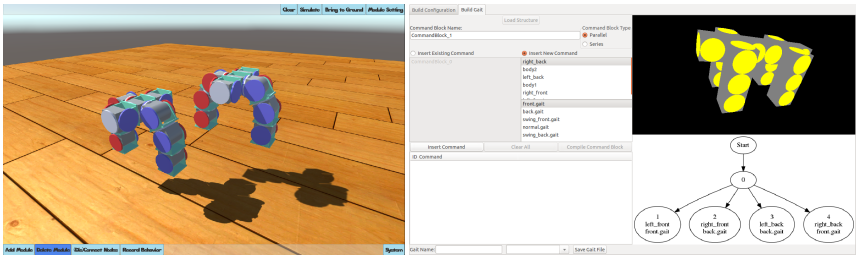

Fig. 6: GUIs for configuration builder (left) and behavior builder (right)

6 Examples and User Perspective

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 in Figure 7. 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 a 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 Figure 7, it is interesting 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.

Figure 8 demonstrates the design flow when a user is designing a configuration and its behaviors. We present the start-to-end user perspective in designing a complicated configuration called Centaur. Consider the order-1 'Chain3" configuration. A second-order "Grasper" configuration, capable of grasping objects, can be formed by combining three first-order "Chain3" configurations. Combining two Graspers allows us to form the legs and body of the third-order "Walkbot" configuration, now using the Grasper arms as legs for walking, as demonstrated in Figure 9-1. If we attach another Grasper to the top of the Walkbot (with two additional modules for structural support), we get the fourth-order "Centaur", a mobile manipulator, as shown in Figure 9-2. Connecting multiple lower-order configurations allows us to quickly develop complex high-order configurations like the Centaur. Given access to a library already containing the Grasper design, for example, creating the Centaur is involves just two composition steps. The user can then immediately compose behaviors already associated with the lower-order configurations (like "Walk" and "Grasp") to create behaviors for higher-order configurations (like picking up and carrying an object).

7 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

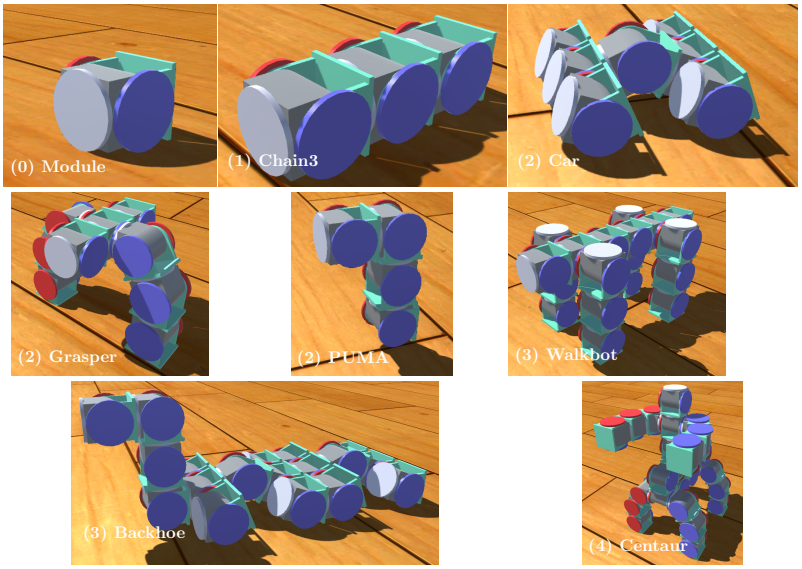

Fig. 7: Library of designs, listed by order. **Order 0:** Module. **Order 1:** Chain3=3x Module. **Order 2:** Car = Chain3 + 4x Module, Grasper = 3x Chain3, PUMA= Chain3 + Module. **Order 3:** Walkbot = 2x Grasper, Backhoe = 2x Car + PUMA. **Order 4:** Centaur = Walkbot + Grasper + 2x Module.

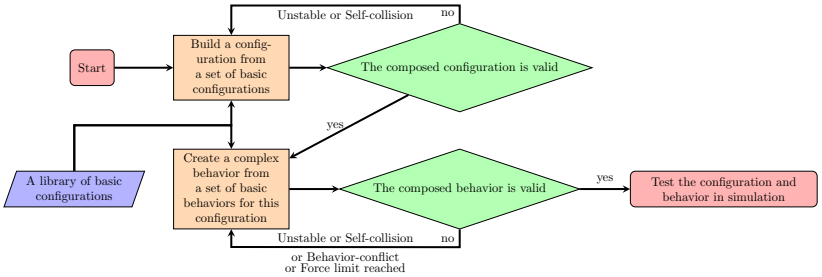

Fig. 8: The design flow

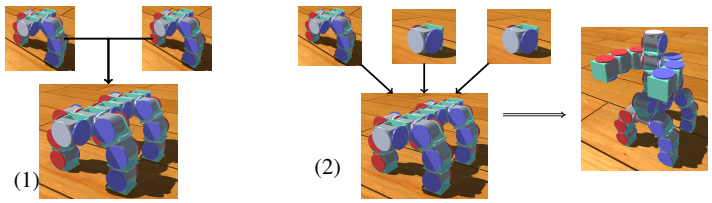

Fig. 9: Building the Centaur. (1) Two Graspers are composed to form a Walkbot. (2) The Walkbot is composed with one more Grasper and two individual modules to form the Centaur.

description language, which allows existing behaviors to be combined in series and parallel to create more complex behaviors. The framework verifies configurations and behaviors, allowing early detection of design flaws, specifically behavior conflict, self-collision, loss of gravitational stability, and forces exceeding the limits of safety for actuators and connectors. In addition to verification, designs can be evaluated in a physical simulator before testing on hardware.

8 Future

Future work will include expansion of the features of our framework. We hope to expand the capabilities of our motion description language without sacrificing ease-of-use, and add more verification tools to assist in problem identification. Another area of future work lies in 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 [14]) to map behaviors into any configuration capable of executing them.

Finally, while our implementation currently supports only SMORES, other modular robots could be easily incorporated. In the future, we plan to incorporate support for the CKbot robot [6] into our software.

References

[1] Physx. http://www.geforce.com/hardware/technology/physx. Accessed: 2015-04-35
[2] Unity3d. http://unity3d.com/. Accessed: 2015-04-35
[3] Bezzo, N., et al.: Demo abstract: Roslaba modular programming environment for robotic applications. In: ICCPS (2014)
[4] Casal, A.: Reconfiguration planning for modular self-reconfigurable robots. Ph.D. thesis, Stanford Univ. (2001)
[5] Christensen, D., Brandt, D., Stoy, K., Schultz, U.P.: A unified simulator for self-reconfigurable robots. In: IROS (2008)
[6] Davey, J., Sastra, J., Piccoli, M., Yim, M.: Modlock: A manual connector for reconfigurable modular robots. In: IROS (2012)
[7] Davey, J., et al.: Emulating self-reconfigurable robots: Design of the smores system. In: IROS (2012)
[8] Fukuda, T., Kawauchi, Y.: Cellular robotic system (cebot) as one of the realization of self-organizing intelligent universal manipulator. In: ICRA (1990)
[9] Hornby, G.S., Lipson, H., Pollack, J.B.: Generative representations for the automated design of modular physical robots. ICRA (2003)
[10] Huang, J., et al.: Rosrv: Runtime verification for robots. In: Runtime Verification, pp. 247–254. Springer (2014)

[11] Kasper Stoy David Brandt, D.J.C.: Self-reconfigurable robots: an introduction. MIT Press (2010)
[12] Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IROS 2004 (2004)
[13] Lipson, H., Pollack, J.B.: Towards continuously reconfigurable self-designing robotics. In: ICRA (2000)
[14] Mantzouratos*, Y., Tosun*, T., Khanna Sanjeev, Y., Mark: On embeddability of modular robot designs. In: ICRA (2014)
[15] Marzinotto, A., Colledanchise, M., Smith, C., Ogren, P.: Towards a unified behavior trees framework for robot control. In: ICRA (2014)
[16] Mehta, A., et al.: A design environment for the rapid specification and fabrication of printable robots. In: ISER (2014)
[17] Pan, J., Chitta, S., Manocha, D.: Fcl: A general purpose library for collision and proximity queries. In: ICRA (2012)
[18] Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)
[19] Salemi, B., Shen, W.M., Will, P.: Hormone-controlled metamorphic robots. In: ICRA (2001)
[20] Sklyarov, V., Skliarova, I.: Design and implementation of parallel hierarchical finite state machines. In: Communications and Electronics, 2008. ICCE 2008. Second International Conference on, pp. 33–38. IEEE (2008)
[21] Sproewitz, A., Moeckel, R., Maye, J., Ijspeert, A.J.: Learning to move in modular robots using central pattern generators and online optimization. The International Journal of Robotics Research **27**(3-4), 423–443 (2008)
[22] Stoy, K., Shen, W.M., Will, P.M.: Using role-based control to produce locomotion in chain-type self-reconfigurable robots. Mechatronics, IEEE/ASME Trans. on (2002)
[23] Valdes, J., Tarjan, R.E., Lawler, E.L.: The recognition of series parallel digraphs. In: Proc. ACM symp. on Theory of comp. (1979)
[24] Walter, J.E., Tsai, E.M., Amato, N.M.: Choosing good paths for fast distributed reconfiguration of hexagonal metamorphic robots. In: ICRA (2002)
[25] Yim, M.: Locomotion with a unit-modular reconfigurable robot. Ph.D. thesis, Stanford (1994)
[26] Yim, M., Duff, D.G., Roufas, K.: Modular reconfigurable robots, an approach to urban search and rescue. In: 1st Intl. Wkshp. on Human-friendly Welfare Robotics Systems (2000)
[27] Yim, M., Duff, D.G., Roufas, K.D.: Polybot: a modular reconfigurable robot. In: ICRA (2000)
[28] Yim, M., et al.: Modular self-reconfigurable robot systems. Robotics & Automation Mag., IEEE (2007)
[29] Yoshida, E., et al.: A self-reconfigurable modular robot: Reconfiguration planning and experiments. IJRR (2002)
[30] Zhang, Y., et al.: Phase automata: a programming model of locomotion gaits for scalable chain-type modular robots. In: IROS (2003)