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

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

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

I. Introduction

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

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

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

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

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

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

II. Contribution and Paper Structure

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

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

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

III. Related Work

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

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

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

While significant progress has been made in the automated generation of modular robot behaviors, automated systems are not yet capable of making modular robots truly useful in practice [10]. The need for new programming techniques to manage the complexity of modular robot systems has been acknowledged in the literature [9]. Historically, gait tables have been a commonly used format in which open-loop kinematic behaviors can be easily encoded [8]. Phased automata have also been presented as a way to easily create scalable gaits for large numbers of modular robots [12]. In this paper, we present a novel scripting language to quickly create complex behaviors for modular robots.

Our framework assists users in verifying design validity by identifying self-collision and loss of gravitational stability. Identification of these conditions is common in modular robot reconfiguration planning [2] and motion planning [11].

IV. Preliminaries

A. SMORES robot

We have developed our system for the SMORES modular robot, developed at the University of Pennsylvania [3]. 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 1). The DoF marked 1, 2, and 4 have rotational axes that are parallel and coincident. Each SMORES module can drive around as a two-wheel differential drive robot. SMORES modules may connect to one another via magnets on each of their four faces, and are capable of self-reconfiguration.

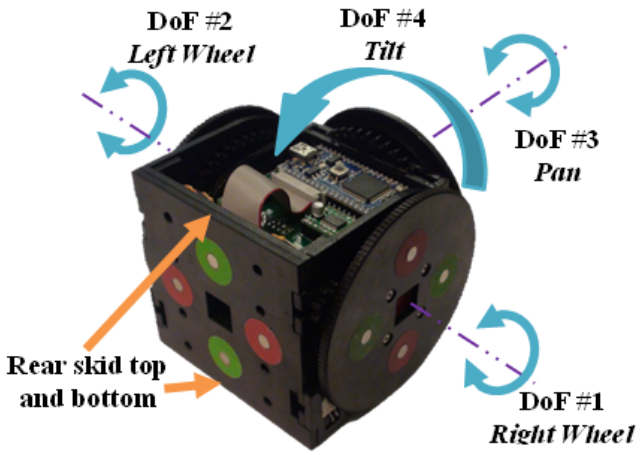


Fig. 1: SMORES robot

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

B. Concepts and terms

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

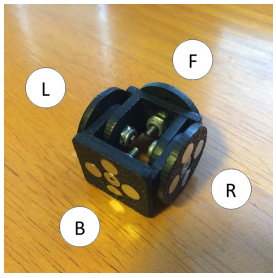
A *behavior* is a programmed sequence of movements for a specific configuration intended to produce a desired effect. A gait for walking is one example. In this paper, we consider open-loop kinematic behaviors represented as series-parallel action graphs, described in detail in section V-B.

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

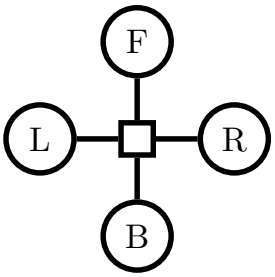
When writing a behavior, it is possible to command one controller to simultaneously hold more than one desired position; this is known as a *controller conflict*. Behaviors with controller conflicts are impossible to execute.

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

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



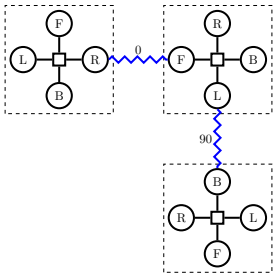
(a) A SMORES module



(b) The graphical representation

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

(a) A configuration



(b) The graphical representation

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

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

V. Approach and Algorithm

TODO: Tarik and Jim write (see subsections)

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

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

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

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

- M is a set of modules, $M = \{\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_q\}$.
- $\mathcal{M}_b$ is single module that will be treated as a base module.
- E is a set of connections between modules. $(\mathcal{M}_i, n_i, \mathcal{M}_j, n_j) \in E$, where $\mathcal{M}_{i,j} \in M, \mathcal{M}_i \neq \mathcal{M}_j$, and $n_{i,j} \in N$.
- $\delta : E \rightarrow \mathbb{R}$ is a labeling function over a connection. Given a connection, δ returns the angle offset in degree between the two connected nodes.

Figure 3b shows a photo of a configuration and its graphical representation. Blue zigzag lines represent connections between modules, and the label of each connection shows the angle offset of that connection.

a) Configuration composition: Define the composition of a set of configurations to a single configuration.

A. Configuration Composition

TODO: Jim writes

b) Input: A set of configurations. A topology graph representing the connectivity among those configurations. A base module (for position transformation).

c) Output: A composed configuration if it is safe.

d) Procedure:

- Start from the configurations that connects to the configuration with base module, transform their positions based on the position of the base configuration and topology graph.
- Check if there is any collisions among the modules and report such collision.
- **Check if the final configuration is stable. If not, find the plane that will make the configuration stable and transform the configuration.**
- Show the expected behavior in simulator.

e) Controller composition: Define the composition of a set of controllers to a single controller. Define the difference between a parallel composition and a series composition. Define the control composition graph.

B. Behavior Representation: 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 fundamental atoms of the language are called actions. An *action* is a tuple (J, X, ξ, T), where J identifies a single DoF of a configuration, X is a controller setpoint for that degree of freedom, ξ specifies an interrupt condition, and T specifies a timeout. Basically, an action defines a setpoint (position or velocity) that the controller will maintain until either the interrupt condition is met or time runs out, whichever comes first. 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). This is similar

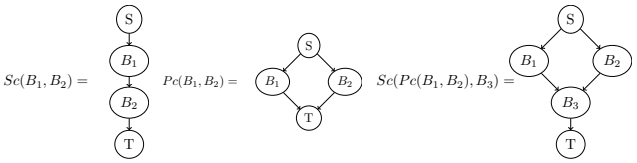


Fig. 4: Series and parallel composition of behaviors

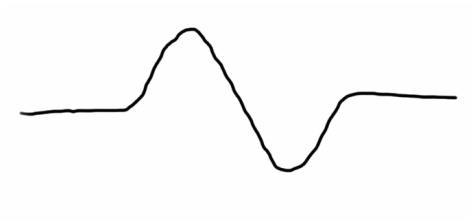


Fig. 5: Zig-zag trajectory **TODO: This is hand-drawn, sorry about that... I will make a more professional-looking replacement**

to typical atomic elements of an MDLe (see [5]) except that our atoms specify controller setpoints for a single DoF, rather than the entire robot. This is because our language includes two fundamental composition operators, P_c (parallel) and S_c (series), to build behaviors by composing sub-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* is 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. B is therefore a *directed series-parallel graph* (SPG). SPG's can always be formed recursively by parallel and series composition operations [**?**]. 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.

Example: Let's say we have the car design shown in Figure X. This design is composed of four wheel modules and a central steering element, which rotates to bend the front wheels to the left or right relative the back, causing the car to turn. The wheel elements expose a function which turns the two side turntables in sync in order to drive forward. The steering element exposes a steering function. We show these behavior definitions below:

$$\mathrm{DRV}(v,t) = Pc\left(\begin{array}{llll}(\ lWheel, & \dot{\theta} = v, & \xi : 0, & T : t\ ),\\ (\ rWheel, & \dot{\theta} = v, & \xi : 0, & T : t\ )\end{array}\right)$$

$$\mathrm{STR}(x) = Pc\left(\begin{array}{llll}(\ tTable1 & \theta = x, & \xi : [\theta = x], & T = \infty\ ),\\ (\ tTable2 & \theta = -x, & \xi : [\theta = -x], & T = \infty\ )\end{array}\right)$$

After composing these sub-configurations into the car, we build higher-level behaviors by composing the behaviors of the sub-configurations. We define functions to go-straight (GST) and turn (TRN):

$$\mathrm{GST}(v,t) = Pc(\ \mathrm{STR}(0),\ \mathrm{DRV}(v,t)\ )$$
$$\mathrm{TRN}(v,t,x) = Pc(\ \mathrm{STR}(x),\ \mathrm{DRV}(v,t)\ )$$

We can now easily define trajectories by sequencing go-straight and turn commands in series. Figure 5 shows the path generated by the zig-zag behavior defined below:

$$\begin{aligned}ZGZ = Sc(&\ \mathrm{GST}(10,5),\ \mathrm{TRN}(10,5,30),\ \mathrm{GST}(10,5),\\ &\mathrm{TRN}(10,10,-30),\ \mathrm{GST}(10,20),\\ &\mathrm{TRN}(10,10,30), \mathrm{GST}(10,5), \mathrm{TRN}(10,10,-30)\ )\end{aligned}$$

C. Controller Composition

TODO: Tarik and Jim write

f) Input: A configurations. A set of controllers. A control composition graph.

g) Output: A composed controller if it is safe.

h) Procedure:

- Compose the set of controllers based on the given control composition graph. Explain how the parallel composition and series composition are handled.
- **Check there is no controller conflict in the composition.**
- Execute the composed controller in user defined incremental time interval. At each time step, update each module position and check collision.
- **At each time step, check if the configuration will not have any unexpected behavior.**

D. Simulation and Verification

TODO: Jim writes

E. Complexity

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

[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.

VI. Example and Experiment

TODO: Tarik and Jim write

With simulation in Gazebo:

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

A. Comparison to Existing Methods

TODO: Tarik writes

In the past, designing configurations and behaviors to address new tasks has required time on the order of one day [7]. **TODO: compare to our results.**

VII. Conclusions

We worked hard, and had fun.

VIII. Future

- How to represent different attribute/ability of the configurations

References

[1] Nicola Bezzo, Junkil Park, Andrew King, Peter Gebhard, Radoslav Ivanov, and Insup Lee. Demo abstract: Roslaba modular programming environment for robotic applications. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pages 214–214. IEEE, 2014.

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

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

[4] Gregory S Hornby, Hod Lipson, and Jordan B Pollack. Generative representations for the automated design of modular physical robots. *Robotics and Automation, IEEE Transactions on*, 19(4):703–719, 2003.

[5] D Hristu-Varsakelis, S Anderson, F Zhang, P Sodre, and PS Krishnaprasad. A motion description language for hybrid system programming. *Institute for Systems Research (preprint)*, 2003.

[6] Ankur Mehta, Nicola Bezzo, Peter Gebhard, Byoungkwon An, Vijay Kumar, Insup Lee, and Daniela Rus. A design environment for the rapid specification and fabrication of printable robots. In *nternational Symposium on Experimental Robotics (ISER), Marrakech, Morocco*, 2014.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.