Science Robotics
AAAS

robotics.sciencemag.org/cgi/content/full/3/23/eaat4983/DC1

Supplementary Materials for

An integrated system for perception-driven autonomy with modular robots

Jonathan Daudelin*, Gangyuan Jing*, Tarik Tosun*, Mark Yim, Hadas Kress-Gazit, Mark Campbell

*Corresponding author. Email: jd746@cornell.edu (J.D.); tarikt@seas.upenn.edu (T.T.); gj56@cornell.edu (G.J.)

Published 31 October 2018, *Sci. Robot.* **3**, eaat4983 (2018)
DOI: 10.1126/scirobotics.aat4983

The PDF file includes:

Text

Other Supplementary Material for this manuscript includes the following:

(available at robotics.sciencemag.org/cgi/content/full/3/23/eaat4983/DC1)

Movie S1 (.mp4 format). Video of demonstrations I, II, and III.

Text

1 Additional Commentary on Related Work

Here we provide a more detailed overview of prior work in MSRR systems. These systems provide partial sets of the capabilities of our system.

The Millibot system demonstrated mapping when operating as a swarm. Certain members of the swarm are designated as "beacons," and have known locations. The autonomy of the Millibot swarm is limited: a human operator makes all high-level decisions, and is responsible for navigation using a GUI [16].

The Swarm-Bots system has been applied in exploration [17] and collective manipulation [18] scenarios. Like the Millibots, some members of the swarm act as "beacons" that are assumed to have known location during exploration. In a collective manipulation task, Swarm-Bots have limited autonomy, with a human operator specifying the location of the manipulation target and the global sequence of manipulation actions.

In [21], Swarm-Bots demonstrate swarm self-assembly to climb a hill. Robots exhibit phototaxis, with the goal of moving toward a light source. When robots detect the presence of a hill (using tilt sensors), they aggregate to form a random connected structure to collectively surmount the hill. A similar strategy is employed to cross holes in the ground. In each case, the swarm of robots is loaded with a single self-assembly controller specific to an *a priori* known obstacle type (hill or hole). The robots do not self-reconfigure between specific morphologies, but rather self-assemble, beginning as a disconnected swarm and coming together to form a random connected structure. In our work, a modular robot completes high-level tasks by autonomously self-reconfiguring between specific morphologies with different capabilities. Our system differentiates between several types of environments using RGB-D data, and may choose to use different morphologies to solve a given high-level task in different environments.

The swarmanoid project (successor to the swarm-bots), uses a heterogeneous swarm of ground and flying robots (called "hand-", "foot-", and "eye-" bots) to perform exploration and object retrieval tasks [19]. Robotic elements of the swarmanoid system connect and disconnect to complete the task, but the decision to take this action is not made autonomously by the robot in response to sensed environment conditions. While the location of the object to be retrieved is unknown, the method for retrieval is known and constant.

Self-reconfiguration has been demonstrated with several other modular robot systems. CKbot, Conro, and MTRAN have all demonstrated the ability to join disconnected clusters of modules together [3, 4, 5]. In order to align, Conro uses infra-red sensors on the docking faces of the modules, while CKBot and MTRAN use a separate sensor module on each cluster. In all cases, individual clusters locate and servo towards each other until they are close enough to dock. These experiments do not include any planning or sequencing of multiple reconfiguration actions in order to create a goal structure appropriate for a task. Additionally, modules are not individually mobile, and mobile clusters of modules are limited to slow crawling gaits.

Consequently, reconfiguration is very time consuming, with a single connection requiring 5-15 minutes.

Other work has focused on reconfiguration planning. Paulos et al. present a system in which self-reconfigurable modular boats self-assemble into prescribed floating structures, such as a bridge [6]. Individual boat modules are able to move about the pool, allowing for rapid reconfiguration. In these experiments, the environment is known and external localization is provided by an overhead AprilTag system.

MSRR systems have demonstrated the ability to accomplish low-level tasks such as various modes of locomotion [1]. Recent work includes a system which integrates many low-level capabilities of a MSRR system in a design library, and accomplishes high-level user-specified tasks by synthesizing elements of the library into a reactive state-machine [7]. This system demonstrates autonomy with respect to task-related decision making, but is designed to operate in a fully known environment with external sensing.

Our system goes beyond existing work by using the self-reconfiguration capabilities of an MSRR system to take autonomy a step further. The system uses perception of the environment to inform the choice of robot configuration, allowing the robot to adapt its abilities to surmount challenges arising from *a priori* unknown features in the environment. Through hardware demonstrations, we show that autonomous self-reconfiguration allows our system to adapt to the environment to complete complex tasks.

2 Library of Configurations and Behaviors

In this work, we use the architecture introduced in [7]. We encode the full set of capabilities of the modular robot, such as driving and picking up items, in a library of robot configurations and behaviors. To create robot configurations and behaviors, users can utilize our simulator toolbox VSPARC (Verification, Simulation, Programming And Robot Construction [5]) presented in [7]. VSPARC allows users to design, simulate and test configurations and behaviors for the SMORES-EP robot system.

Our implementation relies on a framework first presented in [7], which is summarized here. A library entry is defined as $l = (C, B_C, P_b, P_e)$ where:

- C is the robot *configuration*, specified by the number of modules and the connected structure of the modules.
- B_C is a *behavior* that C can perform. A behavior is a controller that specifies commands for robot joints to perform a specific movement.
- P_b is a set of *behavior properties* that describes what B_C does.
- P_e is a set of *environment types* that describe the environments in which this library entry is suitable.

[5] `www.vsparc.org`

To specify tasks at the high level, behavior properties P_b are used to describe desired robot actions without explicitly specifying a configuration or behavior. Environment types P_e specify the conditions under which a behavior can be used. This allows the high-level planner to match environment characterizations from the perception subsystem with configurations and behaviors that can perform the task in the current environment. In Demonstration II, when the environment characterization algorithm reports that the mailbox is located in a "stairs"-type environment, the high-level planner queries the library for configurations that can climb stairs. Since the library indicates that current configuration is only capable of driving on flat ground, the high-level planner opts to reconfigure to the stair-climber configuration, and executes its **climbUp** behavior.

In [7], all robot behaviors are *static* behaviors. That is, once users create a behavior in VS-PARC, joint values for each module are fixed and cannot be modified during behavior execution. Static behaviors, such as a car with a fixed turning radius, do not provide enough maneuverability for the robot to navigate around unknown environments. In this work, we expand the type of behaviors in the library by using *parametric* behaviors, which were first introduced in [23]. Parametric behaviors have joint commands that can be altered during run-time, and therefore allow a wider range of motions. For example, a parametric behavior for a car configuration can be a driving action with two parameters: turning angle and driving velocity. The system associates a parametric behavior with a program that generates values of joint commands based on environment information and current robot tasks. Based on the sensed environment, the perception and exploration subsystem (Section 4.2) can generate a collision-free path, which is used to calculate real-time velocity for the robot. The system then converts the robot velocity to joint values in parametric behaviors at run-time.

To provide an illustrative example, this paper discusses two configurations and their capabilities in detail. The "Car" configuration shown in Figure 2a-5 is capable of picking up and dropping objects in a "free" environment. In addition, the "Car" configuration can locomote on flat terrain. It uses a parametric differential drive behavior to convert a desired velocity vector into motor commands (**drive** in Table 1).

The "Proboscis" configuration shown in Figure 2a-4 has a long arm in front, and is suitable for reaching between obstacles in a narrow "tunnel" environment to grasp objects or reaching up in a "high" environment to drop items. However, the locomotion behaviors available for this configuration are limited to forward/backward motion, making it unsuitable for general navigation.

This library-based framework allows users to express desired robot actions in an abstract way by specifying behavior properties. For example, if a task specifies that the robot should execute a behavior with the **drop** property, the system could choose to use either the Car or Proboscis configurations to perform the action, since both have behaviors with the **drop** property. The decision of which configuration to use is made during task execution, based on the sensed environment. For example, if the perception system reports that the environment is of type "tunnel", the Proboscis configuration will be used, because the library indicates that it can be used in "tunnel"-type environments while the Car cannot.

3 High-Level Planner

In order to generate controllers from high-level task specifications, we first abstract the robot and environment status as a set of Boolean propositions. In Demonstration II, the robot action **drop** is `True` if the robot is currently dropping an object in the mailbox (and `False` otherwise) and the environment proposition **mailBox** is `True` if the robot is currently sensing a mailbox (and `False` otherwise). Moreover the proposition **explore** encodes whether or not the robot is currently searching for the target, the mailbox in this case.

By using a library of robot configurations and behaviors as well as environment characterization tools, we can map these high-level abstraction to low-level sensing programs and robot controllers. As discussed in Section 6, the user specifies high-level robot actions in terms of behavior properties from the library. In Demonstration II, our system can choose to do a drop action by executing any behavior from the library which has the behavior property **drop**, and which also satisfies the current "stairs"-type environment. If the current robot configuration cannot execute an appropriate behavior, the robot will reconfigure to a different configuration that can. In this way, the system autonomously chooses to implement **drop** appropriately in response to the sensed environment. Our system evaluates propositions related to the state of the environment using perception and environment characterization tools in Section 4.2. For example, users can map the proposition **mailBox** to the color tracking function in our perception subsystem, which assigns the value `True` to **mailBox** if and only if the robot is currently seeing a mailbox with the onboard camera. The system treats propositions, such as **explore**, that require the robot to navigate in the workspace differently from the other simple robot actions, such as **drop**. In this example, users can map **explore** to behavior property **drive**, which represents a set of parametric behaviors as discussed in Section 6. In order to obtain joint values for behaviors at run-time, a path planner in the perception and planning subsystem (Section 4.2) takes into account the robot goal as well as the current environment information from the perception subsystem, and generates a collision-free path for the robot to follow. Our system then converts this path to joint values, which are used to execute the **drive** behaviors.

Our implementation employs the Linear Temporal Logic MissiOn Planning (LTLMoP) toolkit to automatically generate robot controllers from user-specified high-level instructions using synthesis [33, 34]. The user describes the desired robot tasks with high-level specifications over the set of abstracted robot and environment propositions that are mapped to behavior properties from the library. LTLMoP automatically converts the specification to logic formulas, which are then used to synthesize a robot controller that satisfies the given tasks (if one exists). The controller is in the form of a finite state automaton, as shown in Figure 7b. Each state specifies a set of high-level robot actions that need to be performed, and transitions between states include a set of environment propositions. Note some propositions are omitted in Figure 7b for clarity. Execution of the high-level controller begins at the predefined initial state in the finite state automaton. In each iteration, LTLMoP determines the values of all environment propositions by calling the corresponding sensing program. Then, LTLMoP chooses the next state in the finite state machine by taking the transition that matches the current value of all environment

propositions. In the next state, for each robot proposition LTLMoP chooses a behavior from the design library which satisfies both the behavior properties and current environment type. For example, in Figure 7b we start in the top state and execute the **explore** program. If the robot senses a mailbox, the value of **mailBox** becomes `True` and therefore the next state is the bottom right state. We then stop the **explore** program and execute the **driveToMailBox** program. We introduce additional constraints to the original task specifications to guarantee that there exist behaviors in the library to implement the synthesized controller. Since self-reconfiguration is time-consuming, the controller chooses to execute the selected behavior using the current robot configuration whenever possible. If the current configuration cannot execute the behavior, the controller instructs the robot to reconfigure to one that can, and if multiple appropriate configurations are available, the controller selects one at random.

4 Reconfiguration

When the high-level planner decides to use a new configuration during a task, the robot must reconfigure. Our system architecture allows any method for reconfiguration, provided that the method requires no external sensing. SMORES-EP is capable of all three classes of modular self-reconfiguration (chain, lattice, and mobile reconfiguration) [22]. We have implemented tools for mobile reconfiguration with SMORES-EP, taking advantage of the fact that individual modules can drive on flat surfaces as described in Section 4.1.

Determining the relative positions of modules during mobile self-reconfiguration is an important challenge. In this work, the localization method is centralized, using a camera carried by the robot to track AprilTag fiducials mounted to individual modules. As discussed in Section 4.1, the camera provides a view of a $0.75\text{m} \times 0.5\text{m}$ area on the ground in front of the sensor module. Within this area, the localization system provides pose for any module equipped with an AprilTag marker to perform reconfiguration.

Given an initial configuration and a goal configuration, the reconfiguration controller commands a set of modules to disconnect, move and reconnect in order to form the new topology of the goal configuration. The robot first takes actions to establish the conditions needed for reconfiguration by confirming that the reconfiguration zone is a flat surface free of obstacles (other than the modules themselves). The robot then sets its joint angles so that all modules that need to detach have both of their wheels on the ground, ready to drive. Then the robot performs operations to change the topology of the cluster by detaching a module from the cluster, driving, and re-attaching at its new location in the goal configuration, as shown in Figure 6. Currently, reconfiguration plans from one configuration to another are created manually and stored in the library. However the framework can work with existing assembly planning algorithms ([31, 32]) to generate reconfiguration plans automatically. Because the reconfiguration zone is free of obstacles, the controller computes collision-free paths offline and stores them as part of the reconfiguration plan. Once all module movement operations have completed and the goal topology is formed, the robot sets its joints to appropriate angles for the goal configuration to

continue performing desired behaviors.

We developed several techniques to ensure reliable connection and disconnection during reconfiguration. When a module disconnects from the cluster, the electro-permanent magnets on the connected faces are turned off. To guarantee a clean break of the magnetic connection, the disconnecting module bends its tilt joint up and down, mechanically separating itself from the cluster. During docking, accurate alignment is crucial to the strength of the magnetic connection [22]. For this reason, rather than driving directly to its final docking location, a module instead drives to a pre-docking waypoint directly in front of its docking location. At the waypoint, the module spins in place slowly until its heading is aligned with the dock point, and then drives in straight to attach. To guarantee a good connection, the module intentionally overdrives its dock point, pushing itself into the cluster while firing its magnets.

Science Robotics
AAAS

robotics.sciencemag.org/cgi/content/full/3/23/eaat4983/DC1

Supplementary Materials for

An integrated system for perception-driven autonomy with modular robots

Jonathan Daudelin*, Gangyuan Jing*, Tarik Tosun*, Mark Yim, Hadas Kress-Gazit, Mark Campbell

*Corresponding author. Email: jd746@cornell.edu (J.D.); tarikt@seas.upenn.edu (T.T.); gj56@cornell.edu (G.J.)

Published 31 October 2018, *Sci. Robot.* **3**, eaat4983 (2018)
DOI: 10.1126/scirobotics.aat4983

The PDF file includes:

Text

Other Supplementary Material for this manuscript includes the following:

(available at robotics.sciencemag.org/cgi/content/full/3/23/eaat4983/DC1)

Movie S1 (.mp4 format). Video of demonstrations I, II, and III.

Text

1 Additional Commentary on Related Work

Here we provide a more detailed overview of prior work in MSRR systems. These systems provide partial sets of the capabilities of our system.

The Millibot system demonstrated mapping when operating as a swarm. Certain members of the swarm are designated as "beacons," and have known locations. The autonomy of the Millibot swarm is limited: a human operator makes all high-level decisions, and is responsible for navigation using a GUI [16].

The Swarm-Bots system has been applied in exploration [17] and collective manipulation [18] scenarios. Like the Millibots, some members of the swarm act as "beacons" that are assumed to have known location during exploration. In a collective manipulation task, Swarm-Bots have limited autonomy, with a human operator specifying the location of the manipulation target and the global sequence of manipulation actions.

In [21], Swarm-Bots demonstrate swarm self-assembly to climb a hill. Robots exhibit phototaxis, with the goal of moving toward a light source. When robots detect the presence of a hill (using tilt sensors), they aggregate to form a random connected structure to collectively surmount the hill. A similar strategy is employed to cross holes in the ground. In each case, the swarm of robots is loaded with a single self-assembly controller specific to an *a priori* known obstacle type (hill or hole). The robots do not self-reconfigure between specific morphologies, but rather self-assemble, beginning as a disconnected swarm and coming together to form a random connected structure. In our work, a modular robot completes high-level tasks by autonomously self-reconfiguring between specific morphologies with different capabilities. Our system differentiates between several types of environments using RGB-D data, and may choose to use different morphologies to solve a given high-level task in different environments.

The swarmanoid project (successor to the swarm-bots), uses a heterogeneous swarm of ground and flying robots (called "hand-", "foot-", and "eye-" bots) to perform exploration and object retrieval tasks [19]. Robotic elements of the swarmanoid system connect and disconnect to complete the task, but the decision to take this action is not made autonomously by the robot in response to sensed environment conditions. While the location of the object to be retrieved is unknown, the method for retrieval is known and constant.

Self-reconfiguration has been demonstrated with several other modular robot systems. CKbot, Conro, and MTRAN have all demonstrated the ability to join disconnected clusters of modules together [3, 4, 5]. In order to align, Conro uses infra-red sensors on the docking faces of the modules, while CKBot and MTRAN use a separate sensor module on each cluster. In all cases, individual clusters locate and servo towards each other until they are close enough to dock. These experiments do not include any planning or sequencing of multiple reconfiguration actions in order to create a goal structure appropriate for a task. Additionally, modules are not individually mobile, and mobile clusters of modules are limited to slow crawling gaits.

Consequently, reconfiguration is very time consuming, with a single connection requiring 5-15 minutes.

Other work has focused on reconfiguration planning. Paulos et al. present a system in which self-reconfigurable modular boats self-assemble into prescribed floating structures, such as a bridge [6]. Individual boat modules are able to move about the pool, allowing for rapid reconfiguration. In these experiments, the environment is known and external localization is provided by an overhead AprilTag system.

MSRR systems have demonstrated the ability to accomplish low-level tasks such as various modes of locomotion [1]. Recent work includes a system which integrates many low-level capabilities of a MSRR system in a design library, and accomplishes high-level user-specified tasks by synthesizing elements of the library into a reactive state-machine [7]. This system demonstrates autonomy with respect to task-related decision making, but is designed to operate in a fully known environment with external sensing.

Our system goes beyond existing work by using the self-reconfiguration capabilities of an MSRR system to take autonomy a step further. The system uses perception of the environment to inform the choice of robot configuration, allowing the robot to adapt its abilities to surmount challenges arising from *a priori* unknown features in the environment. Through hardware demonstrations, we show that autonomous self-reconfiguration allows our system to adapt to the environment to complete complex tasks.

2 Library of Configurations and Behaviors

In this work, we use the architecture introduced in [7]. We encode the full set of capabilities of the modular robot, such as driving and picking up items, in a library of robot configurations and behaviors. To create robot configurations and behaviors, users can utilize our simulator toolbox VSPARC (Verification, Simulation, Programming And Robot Construction [5]) presented in [7]. VSPARC allows users to design, simulate and test configurations and behaviors for the SMORES-EP robot system.

Our implementation relies on a framework first presented in [7], which is summarized here. A library entry is defined as $l = (C, B_C, P_b, P_e)$ where:

- C is the robot *configuration*, specified by the number of modules and the connected structure of the modules.
- B_C is a *behavior* that C can perform. A behavior is a controller that specifies commands for robot joints to perform a specific movement.
- P_b is a set of *behavior properties* that describes what B_C does.
- P_e is a set of *environment types* that describe the environments in which this library entry is suitable.

[5] `www.vsparc.org`

To specify tasks at the high level, behavior properties P_b are used to describe desired robot actions without explicitly specifying a configuration or behavior. Environment types P_e specify the conditions under which a behavior can be used. This allows the high-level planner to match environment characterizations from the perception subsystem with configurations and behaviors that can perform the task in the current environment. In Demonstration II, when the environment characterization algorithm reports that the mailbox is located in a "stairs"-type environment, the high-level planner queries the library for configurations that can climb stairs. Since the library indicates that current configuration is only capable of driving on flat ground, the high-level planner opts to reconfigure to the stair-climber configuration, and executes its **climbUp** behavior.

In [7], all robot behaviors are *static* behaviors. That is, once users create a behavior in VS-PARC, joint values for each module are fixed and cannot be modified during behavior execution. Static behaviors, such as a car with a fixed turning radius, do not provide enough maneuverability for the robot to navigate around unknown environments. In this work, we expand the type of behaviors in the library by using *parametric* behaviors, which were first introduced in [23]. Parametric behaviors have joint commands that can be altered during run-time, and therefore allow a wider range of motions. For example, a parametric behavior for a car configuration can be a driving action with two parameters: turning angle and driving velocity. The system associates a parametric behavior with a program that generates values of joint commands based on environment information and current robot tasks. Based on the sensed environment, the perception and exploration subsystem (Section 4.2) can generate a collision-free path, which is used to calculate real-time velocity for the robot. The system then converts the robot velocity to joint values in parametric behaviors at run-time.

To provide an illustrative example, this paper discusses two configurations and their capabilities in detail. The "Car" configuration shown in Figure 2a-5 is capable of picking up and dropping objects in a "free" environment. In addition, the "Car" configuration can locomote on flat terrain. It uses a parametric differential drive behavior to convert a desired velocity vector into motor commands (**drive** in Table 1).

The "Proboscis" configuration shown in Figure 2a-4 has a long arm in front, and is suitable for reaching between obstacles in a narrow "tunnel" environment to grasp objects or reaching up in a "high" environment to drop items. However, the locomotion behaviors available for this configuration are limited to forward/backward motion, making it unsuitable for general navigation.

This library-based framework allows users to express desired robot actions in an abstract way by specifying behavior properties. For example, if a task specifies that the robot should execute a behavior with the **drop** property, the system could choose to use either the Car or Proboscis configurations to perform the action, since both have behaviors with the **drop** property. The decision of which configuration to use is made during task execution, based on the sensed environment. For example, if the perception system reports that the environment is of type "tunnel", the Proboscis configuration will be used, because the library indicates that it can be used in "tunnel"-type environments while the Car cannot.

3 High-Level Planner

In order to generate controllers from high-level task specifications, we first abstract the robot and environment status as a set of Boolean propositions. In Demonstration II, the robot action **drop** is `True` if the robot is currently dropping an object in the mailbox (and `False` otherwise) and the environment proposition **mailBox** is `True` if the robot is currently sensing a mailbox (and `False` otherwise). Moreover the proposition **explore** encodes whether or not the robot is currently searching for the target, the mailbox in this case.

By using a library of robot configurations and behaviors as well as environment characterization tools, we can map these high-level abstraction to low-level sensing programs and robot controllers. As discussed in Section 6, the user specifies high-level robot actions in terms of behavior properties from the library. In Demonstration II, our system can choose to do a drop action by executing any behavior from the library which has the behavior property **drop**, and which also satisfies the current "stairs"-type environment. If the current robot configuration cannot execute an appropriate behavior, the robot will reconfigure to a different configuration that can. In this way, the system autonomously chooses to implement **drop** appropriately in response to the sensed environment. Our system evaluates propositions related to the state of the environment using perception and environment characterization tools in Section 4.2. For example, users can map the proposition **mailBox** to the color tracking function in our perception subsystem, which assigns the value `True` to **mailBox** if and only if the robot is currently seeing a mailbox with the onboard camera. The system treats propositions, such as **explore**, that require the robot to navigate in the workspace differently from the other simple robot actions, such as **drop**. In this example, users can map **explore** to behavior property **drive**, which represents a set of parametric behaviors as discussed in Section 6. In order to obtain joint values for behaviors at run-time, a path planner in the perception and planning subsystem (Section 4.2) takes into account the robot goal as well as the current environment information from the perception subsystem, and generates a collision-free path for the robot to follow. Our system then converts this path to joint values, which are used to execute the **drive** behaviors.

Our implementation employs the Linear Temporal Logic MissiOn Planning (LTLMoP) toolkit to automatically generate robot controllers from user-specified high-level instructions using synthesis [33, 34]. The user describes the desired robot tasks with high-level specifications over the set of abstracted robot and environment propositions that are mapped to behavior properties from the library. LTLMoP automatically converts the specification to logic formulas, which are then used to synthesize a robot controller that satisfies the given tasks (if one exists). The controller is in the form of a finite state automaton, as shown in Figure 7b. Each state specifies a set of high-level robot actions that need to be performed, and transitions between states include a set of environment propositions. Note some propositions are omitted in Figure 7b for clarity. Execution of the high-level controller begins at the predefined initial state in the finite state automaton. In each iteration, LTLMoP determines the values of all environment propositions by calling the corresponding sensing program. Then, LTLMoP chooses the next state in the finite state machine by taking the transition that matches the current value of all environment

propositions. In the next state, for each robot proposition LTLMoP chooses a behavior from the design library which satisfies both the behavior properties and current environment type. For example, in Figure 7b we start in the top state and execute the **explore** program. If the robot senses a mailbox, the value of **mailBox** becomes `True` and therefore the next state is the bottom right state. We then stop the **explore** program and execute the **driveToMailBox** program. We introduce additional constraints to the original task specifications to guarantee that there exist behaviors in the library to implement the synthesized controller. Since self-reconfiguration is time-consuming, the controller chooses to execute the selected behavior using the current robot configuration whenever possible. If the current configuration cannot execute the behavior, the controller instructs the robot to reconfigure to one that can, and if multiple appropriate configurations are available, the controller selects one at random.

4 Reconfiguration

When the high-level planner decides to use a new configuration during a task, the robot must reconfigure. Our system architecture allows any method for reconfiguration, provided that the method requires no external sensing. SMORES-EP is capable of all three classes of modular self-reconfiguration (chain, lattice, and mobile reconfiguration) [22]. We have implemented tools for mobile reconfiguration with SMORES-EP, taking advantage of the fact that individual modules can drive on flat surfaces as described in Section 4.1.

Determining the relative positions of modules during mobile self-reconfiguration is an important challenge. In this work, the localization method is centralized, using a camera carried by the robot to track AprilTag fiducials mounted to individual modules. As discussed in Section 4.1, the camera provides a view of a $0.75\text{m} \times 0.5\text{m}$ area on the ground in front of the sensor module. Within this area, the localization system provides pose for any module equipped with an AprilTag marker to perform reconfiguration.

Given an initial configuration and a goal configuration, the reconfiguration controller commands a set of modules to disconnect, move and reconnect in order to form the new topology of the goal configuration. The robot first takes actions to establish the conditions needed for reconfiguration by confirming that the reconfiguration zone is a flat surface free of obstacles (other than the modules themselves). The robot then sets its joint angles so that all modules that need to detach have both of their wheels on the ground, ready to drive. Then the robot performs operations to change the topology of the cluster by detaching a module from the cluster, driving, and re-attaching at its new location in the goal configuration, as shown in Figure 6. Currently, reconfiguration plans from one configuration to another are created manually and stored in the library. However the framework can work with existing assembly planning algorithms ([31, 32]) to generate reconfiguration plans automatically. Because the reconfiguration zone is free of obstacles, the controller computes collision-free paths offline and stores them as part of the reconfiguration plan. Once all module movement operations have completed and the goal topology is formed, the robot sets its joints to appropriate angles for the goal configuration to

continue performing desired behaviors.

We developed several techniques to ensure reliable connection and disconnection during reconfiguration. When a module disconnects from the cluster, the electro-permanent magnets on the connected faces are turned off. To guarantee a clean break of the magnetic connection, the disconnecting module bends its tilt joint up and down, mechanically separating itself from the cluster. During docking, accurate alignment is crucial to the strength of the magnetic connection [22]. For this reason, rather than driving directly to its final docking location, a module instead drives to a pre-docking waypoint directly in front of its docking location. At the waypoint, the module spins in place slowly until its heading is aligned with the dock point, and then drives in straight to attach. To guarantee a good connection, the module intentionally overdrives its dock point, pushing itself into the cluster while firing its magnets.