Auton Robot (2015) 38:383–413
DOI 10.1007/s10514-015-9421-8

Distributed reconfiguration of 2D lattice-based modular robotic systems

Ferran Hurtado · Enrique Molina · Suneeta Ramaswami · Vera Sacristán

Received: 12 November 2013 / Accepted: 3 January 2015 / Published online: 17 January 2015

Abstract We prove universal reconfiguration (i.e., reconfiguration between any two robotic systems with the same number of modules) of 2-dimensional lattice-based modular robots by means of a distributed algorithm. To the best of our knowledge, this is the first known reconfiguration algorithm that applies in a general setting to a wide variety of particular modular robotic systems, and holds for both square and hexagonal lattice-based 2-dimensional systems. All modules apply the same set of local rules (in a manner similar to cellular automata), and move relative to each other akin to the sliding-cube model. Reconfiguration is carried out while keeping the robot connected at all times. If executed in a synchronous way, any reconfiguration of a robotic system of n modules is done in $O(n)$ time steps with $O(n)$ basic moves per module, using $O(1)$ force per module, $O(1)$ size memory and computation per module (except for one module, which needs $O(n)$ size memory to store the information of the goal shape), and $O(n)$ communication per module.

Keywords Self-organizing robots · Distributed reconfiguration · Universal reconfiguration

F. Hurtado · E. Molina · V. Sacristán (✉)
Departament de Matemàtica Aplicada II, Universitat Politècnica de Catalunya, Barcelona, Spain
e-mail: vera.sacristan@upc.edu

F. Hurtado
e-mail: ferran.hurtado@upc.edu

E. Molina
e-mail: enrique.molina@estudiant.upc.edu

S. Ramaswami
Department of Computer Science, Rutgers University, Camden, NJ, USA
e-mail: rsuneeta@camden.rutgers.edu

1 Introduction

1.1 Goal

We solve the following problem for 2-dimensional lattice-based modular robotic systems: Given two connected configurations with the same number of modules, reconfigure one into the other by means of a distributed algorithm. To the best of our knowledge, this is the first known general reconfiguration algorithm that applies to both square and hexagonal regular lattices, and uses a general framework that does not exploit specific characteristics of any particular robotic system. Several robotic prototypes currently in existence, as well as several proposed ones, fit within this framework.

More precisely, in our framework, a robot is a connected configuration of homogeneous modules that are located in a 2-dimensional lattice. Each module can attach to and detach from a neighboring module, and can perform some basic movements relative to it. Specifically, it can change its position to a neighboring empty grid position in the lattice by attaching to a neighboring module and moving with respect to it. In addition, we assume each module has constant size memory, can perform constant size computations, and can send or receive constant size messages to or from its neighboring modules. One designated module needs linear memory to store the information of the goal shape and to perform computations required for the reconfiguration algorithm.

Within this framework, our algorithm is completely distributed and local. It consists of a set of rules, each one having a priority, a precondition, and an action or postcondition. Rules are identical for all modules, and are simultaneously executed by all of them. The term "local" here means that each module communicates with and receives information from modules lying within a small neighborhood in order to execute the algorithm. While our algorithm and its

implementation in the simulator are synchronous, we would like to point out that it can be implemented asynchronously at the cost of increased communication between modules. In the procedure we propose, all modules know when they have reached their final destination.

1.2 Related work

Modular self-reconfigurable robotic systems were introduced in the late 1980s. In recent years, they have attracted significant attention from the research community as they are believed to have interesting advantages with respect to fixed-morphology unique-purpose robots. Modular robots are versatile, since they can reconfigure to adapt to different environments and tasks. They are robust, since their units can be interchanged in order to self-repair the system. They are potentially less expensive because the modules can be reused and, in the future, massively produced. As a consequence, self-organizing robotic systems are expected to be used to build emergency structures, repair inaccessible machinery, in outer space missions, and even in current daily life (Yim et al. 2007). One consequence of the flexibility of such robotic systems is the difficulty of planning actions for purposes such as reconfiguration, which is the focus of this work.

We do not attempt to survey the state of the art, as a vast amount of work has been done in recent years to address issues of designing, building, and controlling sets of modules behaving in an autonomous but collaborative way to perform collective tasks. In this section we briefly discuss specific issues and results that are directly related to the results we present in this paper.

Modular robots are frequently classified into homogeneous or heterogeneous, depending on whether their units are all equal or not. Although all units are structurally equal, some units may incorporate or carry special features such as grippers, cameras, antennas, etc. Examples of intrinsically heterogeneous modular robots are I(CES)-Cubes (Ünsal et al. 1999) and AMAS (Terada and Murata 2008). Our work focuses on homogeneous systems. According to the locomotion autonomy of their units, two kinds of self-reconfigurable modular robots can be considered. In the first kind, each unit of the robot has full locomotion capability, such as CEBOT (Fukuda et al. 1992), S-bots (Mondada et al. 2004), and AMOEBA (Liu et al. 2005). In the second kind, locomotion is achieved by cooperation between units, based on the movement of docking joints and links between units. We focus on the latter. Depending on the distribution of the modules in space when connected, self-reconfigurable robotic systems may be organized into lattice, chain, or even hybrid architectures. Examples of chain modular robots are Polypod and its evolution Polybot (Yim et al. 2000), CONRO (Castano et al. 2000), Molecubes (Zykov et al. 2007), and GZ-I (Zhang et al. 2008). Lattice-based modular robots include hexagonal, such as Metamorphic (Pamecha et al. 1996) and Fracta (Murata et al. 1994), triangular, such as Programmable Parts (Bishop et al. 2005), and square or cubic such as Fracta 3D (Murata et al. 1998), vertical (Hosokawa et al. 1998), Crystalline (Rus and Vona 2001), (Butler et al. 2002) and Telecube (Suh et al. 2002), Atron (Jorgensen et al. 2004) or Miche (Gilpin et al. 2008), and rhombic dodecahedral, such as Digital Clay and Proteo (Yim et al. 1997). Probably among the most famous hybrid examples is M-TRAN (Kurokawa et al. 2008), which are modular robots that can behave both as chain and as lattice robots, together with Superbot (Salemi et al. 2006) and, more recently, Roombots (Sproewitz et al. 2009). Our work focuses on lattice-based modular robots.

Many algorithms have been developed for all these robotic systems. Some have just organizing goals, such as finding a leader, detecting holes in the configuration, counting the total number of modules, organizing the robotic system in a tree structure, and other organization tasks (Murata and Kurokawa 2012; Wallner 2009). Some other algorithms have been envisaged to perform tasks involving motion: locomotion, reconfiguration, self-repair, etc. Most of the proposed solutions are centralized algorithms. Recently, however, the need for decentralized and local control has emerged because of the increasing number of modules of the robot (Murata and Kurokawa 2012). Distributed algorithms have been designed for reconfiguring several of the above systems, and more specifically for lattice-based modular robots such as Proteo (Yim et al. 2001), Fracta (Murata et al. 2001; Tomita et al. 1999), Crystalline and Telecube (Butler and Rus 2003; Vassilvitskii et al. 2002; Aloupis et al. 2011) and large scale modular robots such as Catoms (Bhat et al. 2006). It is also worth mentioning the recent growth of interest towards stochastic reconfiguration algorithms (White et al. 2004). Considering hand-coded local rules for reconfiguration a difficult task, (Støy 2006a, b) has proposed a gradient technique for reconfiguring dense objects.

1.2.1 Our work in relation to previous research

Our approach builds on the seminal work of Beni (1988), who proposed the conceptual model of cellular robotic systems, inspired by cellular automata. Some years later, Hosokawa et al. (1998) developed a distributed algorithm for a specific square lattice-based modular robot design, also inspired by cellular automata. It applies the so called sliding cube model, which is comprised of a cube shaped module able to perform three basic moves: slide, convex transition, and concave transition. In Hosokawa et al. (1998), two simple sets of rules are presented for the system to reconfigure from a strip into a staircase and vice-versa. Subsequently, Butler et al. (2004) and Kotay and Rus (2004) proposed a fully decentralized paradigm inspired by cellular automata. In their work, they address locomotion, with and without obstacles, of a

rectangular set of modules, propose rules to reconfigure a strip into a rectangle, and also solve the problem of filling holes in 3D configurations. More recently, Fitch and Butler (2008) proposed a scalable locomotion strategy which is particularly fast for dense configurations. Simultaneously, Dumitrescu et al. (2004a, b), have studied fast locomotion rules for horizontal (vertical) chains and diagonal snake-like formations, and proved universal reconfiguration between 2-dimensional horizontally convex and vertically convex configurations with the same number of modules using local rules, in a linear number of synchronized time steps. Their work also includes theoretical results on decidability for the general case. A similar emphasis has inspired the work of Bojinov et al. (2000), who have proposed specific local rules to produce particular shapes on a 3-dimensional rhombic dodecahedron (Proteo), and (Walter et al. 2004, 2005) for the reconfiguration of some specific configurations of 2-dimensional hexagonal robots. Some years later, in (Ivanov and Walter 2010; Bateau et al. 2012) Walter et al. addressed several issues related to universal reconfiguration, which we also address in our work, albeit in a different setting. More specifically, in their work the authors assume light extra empty space requirements for the movement of the robot units and no communication between them, which leads to synchronous reconfiguration between a class of admissible shapes. Dewey et al. (2008) have also used local rules for a distributed planner in the framework of their general metamodules' theory. Also recently, Kurokawa et al. (2008) have proposed specific sets of rules to produce reconfigurations between particular shapes of M-TRAN which are lattice-based. To the best of our knowledge, their work presents the first execution of a distributed local rules strategy on real robot units, hence proving its realizability beyond experimental simulation. Our work is also related to that of Dumitrescu and Pach (2004), Abel and Kominers (2008), who proved universal reconfiguration for square and cubic lattice-based configurations using the same basic moves, although by means of sequential and centralized algorithms. The novelty of our work is that it is the first universal reconfiguration algorithm (an algorithm that reconfigures any robot shape to any other robot shape with the same number of modules) that is also distributed and local.

1.3 Structure of the paper

In Sect. 2 we describe in detail the framework for our result; namely, the characteristics of the robotic system and the capabilities of its modules, its rule-based behavior, and a description of our syntax for specifying rules. In Sect. 3 we provide an overview of the reconfiguration strategy, which is developed in detail in the following sections. For an easier reading, we first present our results in detail for square lattice-based systems in Sects. 4 and 5. Section 6 is devoted to proving the correctness of our solution, and Sect. 7 analyzes the cost of the reconfiguration algorithm in terms of the number of parallel steps (if run in a synchronous way), the number of moves for each module, and the amount of memory, computation and communication used. Section 8 is devoted to the generalization of our procedures to the hexagonal setting. In Sect. 9, we describe the simulations that we have designed and report on the experimental results. Finally, the paper closes with conclusions and open problems in Sect. 10. An appendix provides some figures related to proofs contained in the text.

2 The model

In this section, we describe the setting for the reconfiguration of modular robotic systems in the square lattice case. See Sect. 8 for its generalization to the regular hexagonal lattice.

In the square lattice setting, a module is any robotic unit located in a 2-dimensional square grid. We represent modules by squares occupying one grid cell, although their actual shape need not be a square (see Sect. 2.1 for examples). A module can independently attach to and detach from each of its 4 direct grid neighboring modules (if present). A robot is a connected set of identical modules. By "connected" we mean that the adjacency graph of the robot configuration (a node in the center of each module and a straight line edge for each attachment among modules) is connected. We say that a robot configuration has no holes if its adjacency graph has no cycles enclosing an empty grid cell in its interior. For robot configurations without holes, the *boundary* is the (possibly self intersecting) closed path formed by all the grid edges in the lattice that have a cell occupied by a module on one side and an unoccupied cell on the other. For configurations with holes, we use the terms *external boundary* and *hole boundary* in the analogous way, and *boundary* for the union of the external boundary and all hole boundaries. See Fig. 1 for an illustration.

Modules cannot move on their own, but they can move relative to each other. To be more precise, we slightly modify the sliding-cube model (Butler et al. 2004), which assumes that a module may perform three relative motions: slide, convex transition, and concave transition (illustrated in Fig. 2a, b, c, respectively. The dark colored module is performing the move.). Our modification consists of introducing a fourth move that we call *opposite transition*: a module that lies between two other modules while connected to one of them may change its connection to the other (see Fig. 2d). The first two moves are of the *change position* type: a module performing *slide* or *convex transition* translates itself from its current lattice position to a neighboring one. The last two moves are of the *change attachments* type: a module performing *concave transition* or *opposite transition* changes its attachment

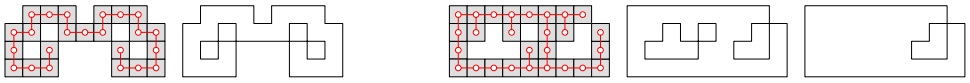


Fig. 1 *Left* a robotic configuration without holes with its adjacency graph and boundary. *Right* a robotic configuration with a hole with its adjacency graph, boundary, and external boundary

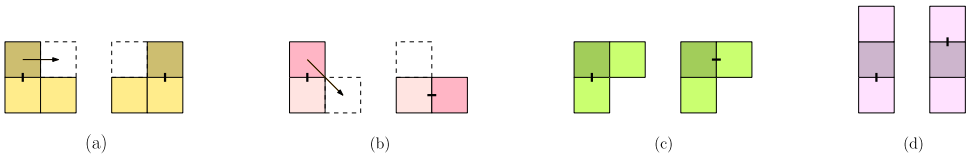


Fig. 2 Change position moves: **a** slide **b** convex transition. Change attachments moves: **c** concave transition **d** opposite transition. All moves may apply in any of the four directions (N, S, E, W) relative to the moving module

Fig. 3 Example of a sliding east rule with a graphical representation (*left*) and its formalization (*right*). The *first line* of the rule contains its name, the *second line* its priority (irrelevant in this very simple example), the *third line* contains the precondition, and the *last line* the action or postcondition. In the graphical representation, the active module applying the rule is depicted in a *darker color*

from one neighbor to another without modifying its lattice position.

In our framework, the modules of the robot are indistinguishable, and each module is given and applies the same set of rules. In order to do so, we assume each module has a (simple) processor and some (small) memory, knows its own orientation (N, S, E, W) and state (a short text string), can detect whether it is attached to a neighbor, can send and receive (short) messages to and from neighbors, and is able to perform (elementary) operations with a few counters and text strings. For our reconfiguration algorithm, only one module needs to store the final configuration, which is a linear amount of information. This module, that we call the leader, can be either determined in advance or autonomously chosen by the set of modules (Nichitiu et al. 2001; Wallner 2009). See Rodríguez (2013), Ordóñez (2013) for a detailed description of the requirements implemented in the actual simulation system.

As stated above, all modules run the same predefined set of rules. Each rule has the following structure: a priority, a precondition, and an action or postcondition. Priorities, represented as small integers, are used by the module to decide which of possibly several rules that apply to its situation is executed. A precondition is any constant size boolean combination of the following: compare priorities, check neighboring empty/filled positions, check own connections, match states/text or counters/integers, and compare calculation results with counters, messages and integers. A postcondition can be any *and* combination of the following: change position (slide, convex transition), change attachments (concave transition, opposite transition), modify state, compute and update counters, and send messages. Figure 3 shows a simple example.

2.1 Prototypes

In this section, we briefly analyze the extent to which the abstract model proposed in Sect. 2 represents current and proposed prototypes of modular robotic systems.

As pointed out in (Butler et al. 2004), the sliding cube model can be instantiated by several current prototypes, either by their atomic robot units or by means of meta-modules. This is also the case for our extended model. In fact, the addition of the opposite transition move does not restrict the class of robotic systems that can perform it, since the sliding cube model already assumes that each module can attach to and detach from any of its neighboring ones.

Our version of the convex transition move, however, only assumes that the goal grid position is empty. It does not require any empty space between the current and the goal position of the moving module, as it does, for example, in the models considered in Butler et al. (2004), Dumitrescu and Pach (2004), Benbernou (2011). This assumption is important because otherwise some configurations are blocked and cannot be reconfigured (see Fig. 1, left), whereas in our model, we prove universal reconfiguration, i.e., reconfiguration between any two shapes with the same number of modules.

Our assumption about the convex transition move could be a potential limitation because the atomic robot units of several current prototypes need some extra empty space to produce convex transitions. For example, in the sliding model

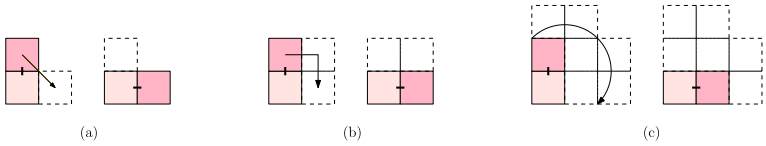


Fig. 4 **a** Our model, **b** the sliding model. **c** the rotating model. In all cases, the grid cells required to be empty are depicted in *white*

presented in Hosokawa et al. (1998), Chiang and Chirikjian (2001), An (2008), the intermediate lattice cell between the initial and the goal positions of the moving module needs to be empty; see Fig. 4b. In the rotating model (Yoshida et al. 2000; Ünsal et al. 1999), even more space needs to be free to avoid collisions of the moving module with the static ones; see Fig. 4c.

There are some prototypes, however, that fit our model. Molecule (Kotay and Rus 2000) and lattice configurations of M-Tran (Kurokawa et al. 2008) can use the third dimension to produce a 2-dimensional move. The metamorphic robot (Pamecha et al. 1996) is a clear example of a hexagonal prototype which can also perform our convex transition move.

Grouping atomic robot units into meta-modules also ensures that convex transition moves can be safely made without extra free space requirements. Some of the most well known examples of robots that display such behavior are crystalline and telecube robots (Rus and Vona 2001; Butler et al. 2002; Suh et al. 2002). By means of their expand/compress capability, metamodules of these robotic systems, made out of 2×2 atoms, can compress such that two metamodules occupy one single lattice cell. Hence, referring to Fig. 2a, the top module can perform a convex transition by first compressing into the bottom module and then uncompressing to the right. In Aloupis et al. (2013), it was proven that other current prototypes, such as M-Tran (Kurokawa et al. 2008) and Molecube (Zykov et al. 2007) can behave like Crystalline or Telecube meta-modules, which implies that they can instantiate our general model. In fact, the authors of Aloupis et al. (2013) claim that many other robotic systems, such as PolyBot (Yim et al. 2000), SuperBot (Salemi et al. 2006), Roombots (Sproewitz et al. 2009), and Atron (Jorgensen et al. 2004) have the same behavior when organized in the appropriate meta-modules.

In the sliding model, it is easy to prove that the meta-modules depicted in Fig. 5a allow slide and convex transition moves without the need for extra free space. See Figs. 39 and 41 in the Appendix for a partial depiction of how these meta-modules achieve the slide transition, and Figs. 40 and 42 for convex transition. The case of the rotating model also allows both sliding and convex transitions without using extra space if units of the meta-modules are only required to be connected through their vertices. Figure 5b shows one possible meta-module for this case and Figs. 43 and 44 in the Appendix show the meta-module moves that achieve slide and convex transition without extra space. In fact, for both

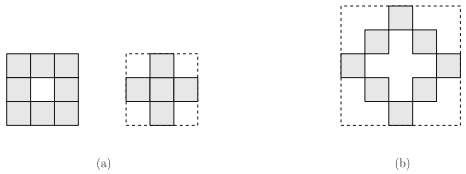


Fig. 5 **a** Two possible meta-modules of sliding units. **b** A possible meta-module of rotating units. In both cases, slide and convex transition can be obtained without extra space and by maintaining the component units of each meta-module

of the proposed meta-modules, slide and convex transition moves can be performed without any exchange of atomic units between the meta-modules involved, as is illustrated in the Appendix.

3 Overview of reconfiguration strategy

In this section, we present a high-level overview of the strategy employed to achieve universal reconfiguration of 2-dimensional lattice-based modular robots by means of a distributed algorithm. The solution we present is distributed because each module acts on its own without the need of a central controller, other than to get the reconfiguration process started. The starting command may be broadcast to all modules or sent to just one module, from where it can be transmitted to the entire set via the connectivity of the robot. Our solution is parallel as all modules act in parallel. In fact, when running the simulations, it becomes evident that our strategy allows many modules to act simultaneously because it is local (see Sect. 9 for details). The reconfiguration rules are designed to prevent conflicts such as collisions between modules or obstructions created by modules trapped in bottlenecks. Our solution is local because each module only needs to communicate with modules within a small neighborhood when checking rule preconditions, such as testing for empty/full grid positions, comparing counter values in neighboring modules, etc. In this context, the neighborhood of a module consists of all modules lying in grid positions within the second annulus around it (see Figs. 13, 14 for examples). The only way to eliminate the need for communication between modules within distance two of each other is to restrict the shape and connectivity of the robots being reconfigured. Note that the need for a module to communicate with another module within the second annulus is necessary only to prevent collisions; that is, only collision rules use this form of inter-module communication.

The overall strategy behind our algorithm is to move modules along the boundary of the robot to reconfigure in two stages. We first reconfigure the robot from its initial shape into a canonical shape (the strip configuration, in our case) and then reconfigure from the canonical to the final shape. The modules do not need to know the robot's complete initial shape. However, the goal shape needs to be known at least by the leader, which is a specially designated module. In particular, our solution to reconfigure from the canonical to goal shape requires the leader to assign a final destination location for each module in the canonical configuration.

Our solution is based on the following general operating principles:

1. A particular spanning tree of the robot's adjacency graph, which we call the *scan tree*, is built so that all leaves of the tree lie on the boundary of the robot. At the beginning, all modules are considered to be static. At any given instant, only leaf modules can start moving, i.e., go from static to active. Once a module is active, its node is considered to be cut from the spanning tree.
2. The movement of the modules along the boundary of the robot always follows the right hand rule (turn right along the robot boundary) when reconfiguring from the initial to canonical shape, and the left hand rule when reconfiguring from the canonical to goal shape.
3. Moving modules are not allowed to climb (move relative to) other moving modules. This is a reasonable assumption if we want to avoid unbounded acceleration and unpredictable collisions.
4. Every module is assigned a number when constructing the above stated spanning tree. Generally speaking, this number corresponds to the DFS (depth first search) order numbering of the nodes of the scan tree of the initial shape for the initial to canonical reconfiguration, or the goal shape for the canonical to goal reconfiguration. This number is used to guide the moves of the modules and also to prove the correctness of our solution.

The next two sections provide details for each of the two stages of our reconfiguration algorithm.

4 Forward reconfiguration: initial to canonical

In Sect. 4.1, we first describe the preprocessing steps that initialize each robot module with the preliminary data that it needs to carry out the moves that reconfigure the robot from the initial shape to the canonical strip configuration. The details of the moves themselves, such as the rules required to activate and advance the modules, avoid collisions and obstructions, and finally place the modules in a strip configuration, are provided in Sect. 4.2.

4.1 Preprocessing

Our algorithm requires a specially designated node, which we call the *leader*, that serves multiple purposes like being responsible for "waking up" all the modules, serving as the root node for the DFS numbering of the nodes (refer to item 4 in Sect. 3), and being the first node in the strip configuration. Section 4.1.1 describes how we choose the leader.

In order to build the scan tree for the robot (refer to item 1 of the previous section), it is necessary to detect holes in the robot. Section 4.1.2 describes how this is done. The procedure for building the scan tree itself is given in Sect. 4.1.3. Finally, the procedure to find the DFS number for each node is described in Sect. 4.1.4.

4.1.1 Choosing a leader

For the construction of the canonical configuration, the leader needs to lie on the external boundary of the configuration (the reason for this requirement will become evident in Sect. 4.2). For the remainder of this paper, we assume that the leader is the topmost among the rightmost modules in the configuration. We also assume that every module is in an initial sleep state, i.e. `State=sleep`, prior to starting the reconfiguration.

The leader module may be either designated by an external controller or chosen by the modules themselves (Nichitiu et al. 2001). In the case of the former, the leader simply prepares the modules for the next preprocessing step, which is to detect all the holes in the robot. This is done by propagating a message from the leader to set `State=detect_holes` in every module that has `State=sleep`. In the case of the latter, the broadcast message is intended to change each module's state into `State=choose_a_leader` so that the rules described in Wallner (2009), Rodríguez (2013), Ordóñez (2013) apply and the modules choose the leader on their own. At the end of this process, every module is aware that a leader is chosen. As in the other case, the leader now propagates a message to set `State=detect_holes` in all modules.

4.1.2 Detecting the holes

The next step is to recognize whether or not the robot configuration has holes and, if it does, to detect the leftmost among the bottommost modules of each hole boundary. For the rest of this paper we will call such a module an *lbh-module*. This can be done by means of local rules very similar to those described in Wallner (2009) for choosing a leader along the boundary. See Rodríguez (2013), Ordóñez (2013) for details. The result is that each module in the configuration has the information of whether or not it is an lbh-module and the new state for all modules is `State=build_scan_tree`.

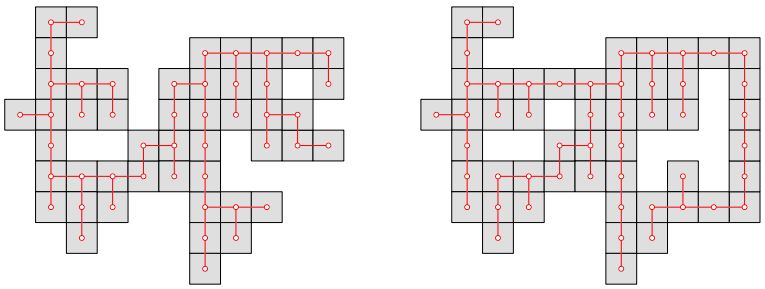


Fig. 6 *Left* the scan tree of a configuration without holes. *Right* the scan tree of a configuration with holes

```
Build scan tree East-South
1
State=build_scan_tree & blh=true
Tree_neighbor(1,0) & Tree_neighbor(0,-1) & State=numbering
```

```
Build scan tree East-West-South
1
State=build_scan_tree & blh=false & Position(0,1)=empty
Tree_neighbor(1,0) & Tree_neighbor(-1,0) & Tree_neighbor(0,-1) & State=numbering
```

```
Build scan tree North-South
1
State=build_scan_tree & Position(0,1)=full
Tree_neighbor(0,1) & Tree_neighbor(0,-1) & State=numbering
```

Fig. 7 Rules for building the scan tree

In our simulation, we implement both procedures to choose a leader and detect holes (see Sect. 9), and in fact, they are carried out simultaneously in our implementation.

4.1.3 Building the scan tree

Our reconfiguration algorithm uses a particular spanning tree of the adjancency graph of the robot, which we call the *scan tree*. The scan tree is a graph in which there is a (vertical) edge between every pair of vertically adjacent modules. In addition, each vertical column of connected modules is attached to its neighboring vertical columns by (horizontal) edges corresponding to the uppermost horizontally adjacent pair of modules. When the robot configuration has no holes, each vertical column of connected modules consists of exactly one connected component and hence it is trivial to see that the resulting graph is connected and acyclic. However, when the configuration has holes, each vertical column of connected modules could have multiple connected components. As a result, the addition of horizontal graph edges creates cycles around each hole. We rectify this situation by removing the horizontal edge between a hole's lbh-module and its neighbor to the west. This breaks all cycles, resulting in a spanning tree of the robot's adjacency graph. Figure 6 shows two simple examples.

The scan tree is formed by means of the rules illustrated in Fig. 7. `Tree_neighbor` establishes a scan tree adjacency with another module (given in relative coordinates). For example, `Tree_neighbor(0, -1)` makes the module to the south a scan tree neighbor. Observe that the first rule in Fig. 7 ensures acyclicity of the scan tree in robots with holes.

In a physical robot, it would be preferable to separate the logical structure of the scan tree from the physical connectivity between modules for the sake of greater physical stability. If two modules are not adjacent in the scan tree, only the logical link between them is missing while the physical link stays. However, in our implementation (see Sect. 9), we start with a configuration in which all attachments among adjacent modules are present, and then detach modules that are not adjacent in the tree. We do this to make the structure of the tree clearer during simulations.

4.1.4 Numbering the modules

Once the tree has been built, a positive integer value is assigned to each module. The assigned numbers, referred to as *DFS numbers*, correspond to the sequential numbering of the modules in a counterclockwise depth-first traversal of the scan tree starting at the leader node. The DFS number of each module is computed by starting at the leader and spreading its value by following the scan tree edges. The leader sets its DFS number `Num` to zero and sends the value to its first scan tree neighbor in counterclockwise order. When a module M receives the DFS number from a neighboring module and does not yet have its own `Num`, it increments the DFS number by 1 and sets its `Num` to that value, and then passes it on to its first scan tree neighbor in counterclockwise order. If no such neighbor exists (because M is a leaf node of the scan tree), `Num` is passed back to the parent. If

Fig. 8 Rules for assigning DFS numbers

```
Start numbering
1
Leader=true & State=numbering & Num={}
Num=0 & Send 1 to next child & Mark child
```

```
Spread numbering first
1
State=numbering & Num={} & Received n from neighbor & Not all neighbors are marked
Num=n+1 & Send n+1 to next child & Mark child
```

```
Spread numbering continue
1
State=numbering & Num≠{} & Received n from neighbor & Not all neighbors are marked
Send n to next child & Mark child
```

```
Spread numbering last
1
State=numbering & Num≠{} & Received n from neighbor & All neighbors are marked
Send n to parent (if any) & State=to_strip
```

M already has its own Num, it simply passes on the DFS number to its next unvisited scan tree neighbor if it exists, or back to the parent if it does not. The process is illustrated in Fig. 8.

Simultaneously with the DFS number Num, the modules also compute two other functions, namely Min and Max, which are related to Num. For the sake of clarity, imagine that Num has already been assigned to each module. In the scan tree, two nodes of degree greater than two are said to be consecutive if the unique path between them contains only nodes of degree two. This maximal path of degree two nodes is referred to as a *branch*. For each node a in the tree, Min is the DFS number of the the first module in its branch, and Max is one more than the largest DFS number assigned in the subtree rooted at a. Observe that all nodes in a branch have the same Min and Max numbers. It is not difficult to see that Min and Max can be computed along with Num during the depth first traversal of the scan tree. The value of Min at each node is established when the node is traversed for the first time: a node of degree greater than two has Min equal to its Num and a node of degree two or less receives its Min value from its parent. Similarly, Max is established when a node is traversed for the last time: a leaf node has Max equal to its Num and a non-leaf node receives its Max value from its child. The rules of Fig. 8 can be easily modified to simultaneously find Num, Min, and Max. See Ordóñez (2013), Rodríguez (2013) for more details, and Fig. 9 for an example.

Notice that the value of Max at the leader indicates the total number of modules in the configuration. Min and Max allow us to identify modules belonging to the same branch, and also to establish order between branches in the depth first traversal. Num, Min, and Max will be used throughout our reconfiguration algorithm.

4.2 Reconfigure to canonical strip

After the scan tree is constructed and the values Num, Min, and Max have been computed by all modules, the reconfiguration may start. The initial robot configuration is transformed into a canonical shape via relative movement of the modules. To keep the explanation simple, the canonical shape is a horizontal strip lying to the right of the leader. Recall that we choose the leader as the rightmost and topmost module, making this canonical configuration feasible. It is easy to modify the rules to produce any other fixed shape that lies to the right of the leader.

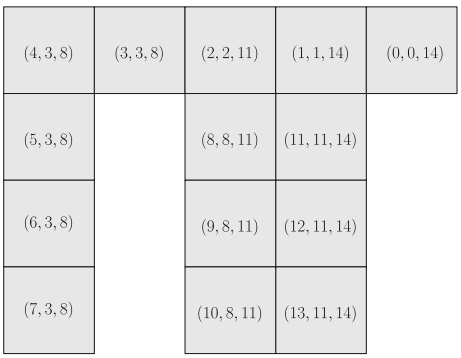


Fig. 9 Example of a numbered configuration. For each module, the triple (Num, Min, Max) is shown

The reconfiguration is done by means of three sets of rules: activation rules (Sect. 4.2.1), advance rules (Sect. 4.2.2), and canonical strip rules (Sect. 4.2.4). In this section we describe the rules, but we do not attempt to justify their correctness, which is shown in Sect. 6.

4.2.1 Activation rules

At any given stage of the reconfiguration algorithm, some modules are static and some are active. At the beginning, all modules are static (State=to_strip). The tree of static modules is called the *static tree*. During the reconfiguration, any leaf of the static tree that is not attached to an active module becomes an active module (State=forward). An active module never reverts to static. Once a module becomes active, it is pruned from the static tree. While the static tree

```
Activate leaves
1
State=to_strip & Leaf=true & No active module attached
State=forward
```

Fig. 10 Activating leaves

gets pruned during the reconfiguration, it always remains a scan tree of the static modules. For our reconfiguration algorithm, DFS numbers are relevant only for static modules; consequently, DFS numbers are not shown for active modules in the figures. Activation is achieved by means of the rule illustrated in Fig. 10.

4.2.2 Advance rules

Active modules use the four allowed moves—slide, convex transition, concave transition, and opposite transition - based on the following principles:

- Advance follows the right hand rule along the boundary of the static tree.
- Advance moves are always made relative to static modules. The static modules on which active modules are moving is called the *substrate*.
- A module advances from one position with substrate module a to another with substrate module b only if $\texttt{Num}(a) \geq \texttt{Num}(b)$. See Fig. 11 for an illustration. Intuitively, this rule ensures that an advance move always gets a module closer to its final destination, i.e., moves it towards the leader.

Advance rules are designed to take care of all possible conflicts:

1. *Activation conflicts* occur when an active module tries to move to a position in which it attaches to a static leaf that simultaneously becomes active. In this case, priority may be given to the leaf activation or the moving module. Either choice is appropriate, as long as it stays consistent during the reconfiguration. Activation conflicts are easy to avoid by simply adding a precondition to all activation rules (in case of the former choice) or all moving rules (in case of the latter choice).
2. *Collision conflicts* occur when two active modules intend to move to the same grid cell. In this case, priority is given to the module whose substrate module has a lower DFS number. Specific rules implementing this priority are described below.
3. *Obstruction conflicts* occur when an active module tries to move into a grid cell which is already occupied by an active module. In this situation, we have two possible cases. In the first case, the obstructing module is advancing in the same direction as the obstructed module, just one step in front. In this event, the obstructed module waits. In the second case, the two modules are advancing in opposite directions along portions of the boundary that are just far enough apart to produce a bottleneck. A module entering a bottleneck and one exiting it would obstruct each other and halt the reconfiguration. Bottleneck conflicts are handled by means of special rules called *jumping rules*.

We now describe the precise rules that produce the behavior described above. The east concave transition rule is illustrated in Fig. 12. Since concave transitions produce no movement of the module from one grid cell to another, there are no collision or bottleneck issues to address. As an example, we include the actual syntax of the rule.

Figure 13 illustrates the rules required for eastward slide; the precondition is implied by the legends on the dashed grid positions. In the rule depicted on the left, the precondition guarantees no collisions. It is implicitly understood that the right module (with DFS number k) is static and will remain that way during the transition. In the rule depicted on the right, there is a possible collision, in which case the module whose substrate module has lowest DFS number moves. Notice that there are no other situations that may cause collision.

Rules for convex transition are analogous to those for slide. They are illustrated in Fig. 14. On the left, the precondition guarantees no collisions. On the right, the module whose substrate module has lowest DFS number moves in case of a conflict.

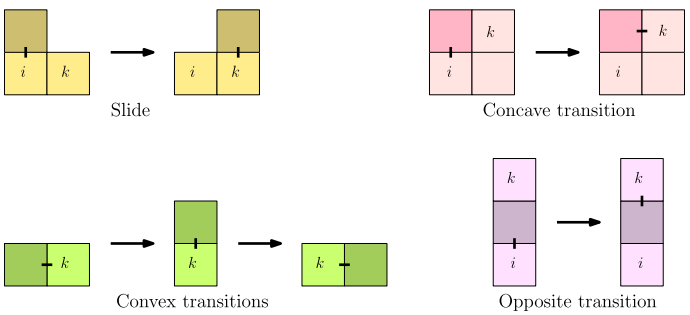


Fig. 11 *Legends* on modules indicate their DFS numbers. All transitions take place only if $k < i$. *Light* modules are static, *dark* modules are active

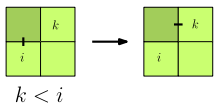


Fig. 12 The east concave transition rule. Legends on modules indicate their DFS numbers (Color figure online)

```
Concave transition to east
2
State=forward & Attached(0,-1) & State(1,0)=State(1,-1)=static & Num(1,0)<Num(0,-1)
Attach(1,0) & Detach(0,-1)
```

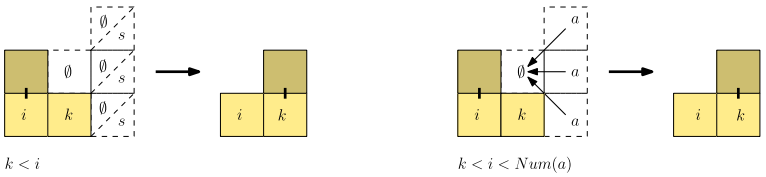


Fig. 13 Two eastward sliding rules. Legends on *shaded* modules indicate their DFS numbers. Legends on *dashed grid* positions read as follows: ∅ means that the cell is empty, *s* means that it is occupied by a static `to_strip` or `wait_in_strip` module, and *a* means that it is occupied by an active `forward` module. The *arrows* show the intended final destination of the potentially colliding active modules (Color figure online)

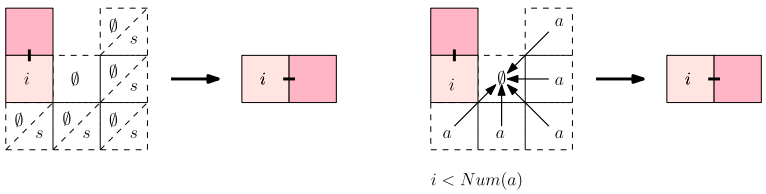


Fig. 14 Two south-east convex transition rules. Meanings of *symbols* are explained above in Fig. 13 (Color figure online)

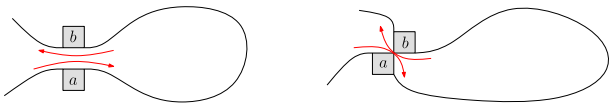


Fig. 15 Two possible bottleneck types created by a pair of co-vertical modules (*left*) or a pair of modules sharing a vertex (*right*). *Arrows* indicate the directions in which modules may attempt to move through the bottleneck

The priority of these advance rules is equal to 2 so that solving bottlenecks has priority over advancing. In the following sections, we describe the jumping rules designed to avoid bottlenecks. All such rules have priority 1. In a jumping rule, an active module checks whether it is entering a bottleneck prior to executing an advance move of any sort (slide, convex transition, or concave transition). If so, it changes attachments to jump ahead along the boundary, which is equivalent to exiting the bottleneck. The following section describes bottleneck rules in detail.

4.2.3 Avoiding bottlenecks

In our model, a robot configuration may have two types of bottlenecks. One is created by two modules that are co-horizontal or co-vertical (along the grid) and separated by an empty grid cell. See Fig. 15 (left). The other is created by two modules sharing a vertex. See Fig. 15 (right).

Let a and b be two modules in a bottleneck configuration, as shown in Fig. 16. Let x be an active module attached to a (say). Observe that x must trap a portion of the robot's boundary on one side, creating a *cul-de-sac*. Since x is advancing along the boundary towards the leader by following the right hand rule, x may be either entering the cul-de-sac [Fig. 16 (left)] or exiting the cul-de-sac [Fig. 16 (right)]. Note that the presence of module x in the bottleneck raises the possibility of a deadlock of active modules resulting from an obstruction conflict, a situation in which no module is able to move. In order to avoid such a conflict, any active module located in the entrance of a cul-de-sac is obliged to *jump* to the exit of the cul-de-sac by changing attachments from one of the modules (say a) of the bottleneck configuration to the other (b). Observe that, in order to distinguish entrance from exit we need to infer global information (namely, the side of the bottleneck that contains the cul-de-sac) from local information (namely, the modules neighboring x). The function `Max` is used precisely for this purpose.

Topological analysis for jumping rules Let a and b, with $\texttt{Num}(a) > \texttt{Num}(b)$, be two modules in a bottleneck configuration, and let module c be the least common ancestor of a and b in the DFS tree. Then two possible cases arise: (1) c is distinct from a and b, in which case the path from c to b is traversed before the path from c to a during the DFS traversal, or (2) c is not distinct from a and b, in which case c

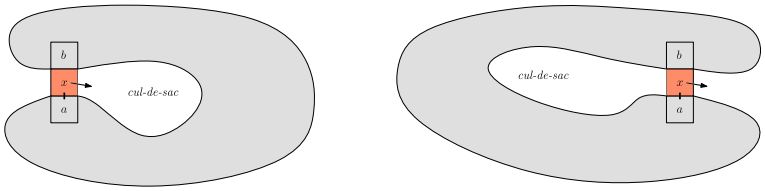


Fig. 16 Active module x may be entering (*left*) or exiting (*right*) the cul-de-sac

must be identical to b. We now consider each of these cases in detail.

Case 1. c is distinct from a and b, as illustrated in Fig. 17.

Since the path from c to b is traversed before the path from c to a, there are four possible configurations for these paths, as shown in Fig. 17. Suppose that module x is attached to module a. Observe that in configuration 1a and 1c, x must be entering the cul-de-sac because the DFS numbers of modules in the path from a towards c are decreasing. In configurations 1b and 1d, x must wait to advance because the DFS numbers of modules from a to the end of the branch are increasing. Hence, in all four configurations, x avoids entering the cul-de-sac by changing its attachment from a to b. In cases 1a and 1b, x will not be able to advance as soon as it changes attachments because the DFS number increases in front of it. However, this will not produce a permanent obstruction, since these modules are not trapped in the cul-de-sac. Whenever the modules of this branch activate and start moving, x will be free to advance as well. After that, any modules trapped in the cul-de-sac (such as the ones ahead of module a in case 1b, for example) will eventually advance. In cases 1c and 1d, once module x changes attachments from a to b, it will be able to advance along the branch of b.

Case 2. c is identical to b, as illustrated in Fig. 18.

Since x follows the right hand rule when moving along the boundary, changing attachments from a to b would ensure that module x does not enter the cul-de-sac in cases 2a and 2b, whereas changing attachments from b to a would ensure that module x does not enter the cul-de-sac in cases 2c and 2d. Observe also that x can keep advancing after the jump in cases 2a, 2b, and 2d. In case 2c, though, x will not be able to advance as soon as it changes attachments because the DFS number increases in front of it. Nevertheless, this will not produce a permanent obstruction, as the modules in front of x are not trapped in the cul-de-sac: they will eventually activate and move, and then x will be free to advance as well. Notice that jumping rules are the only rules that allow an active module to change attachments from a lower DFS number to a higher one.

Implementation of jumping rules As mentioned previously, a jump move has priority over an advance move: every time an active module x finds itself attached to a static module (a or b) and neighboring another static module (b or a), it checks whether or not the situation warrants a jump. This check requires the use of functions `Num` and `Max`. Note that Case 1 holds (that is, b and a have a lowest common ancestor that is distinct from both) iff $\texttt{Max}(b) < \texttt{Num}(a)$. Cases 2a and 2b are distinguished from cases 2c and 2d as follows: Module b is the lowest common ancestor of a and b iff $\texttt{Num}(b) < \texttt{Num}(a) \leq \texttt{Max}(b)$. Furthermore, we consider the module b' that is the first neighbor of b along the boundary as we go in the clockwise direction from b to a (refer to Fig. 18). Note that $\texttt{Num}(b') > \texttt{Num}(b)$ for Cases 2a and 2b, whereas $\texttt{Num}(b') < \texttt{Num}(b)$ for Cases 2c and 2d. Therefore, a few comparisons of the values of functions `Num` and `Max` in modules a, b, and b' allow module x to determine whether or not to change attachments from a to b or from b to a.

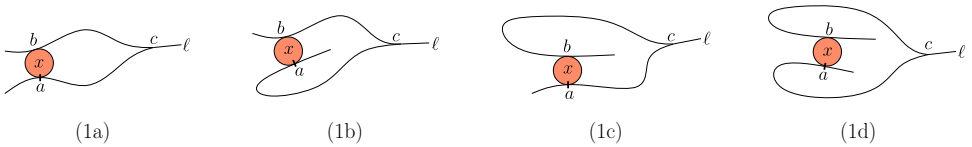


Fig. 17 The four possibilities in a bottleneck configuration if c is distinct from a and b. In all cases, module x applies a jumping rule, i.e., it changes attachments from a to b. The *letter* ℓ indicates the topological position of the leader of the configuration

Fig. 18 The four possibilities in a bottleneck configuration if c is identical to b. The letter ℓ indicates the topological position of the leader of the configuration. Recall that b must have lower DFS number than a. This implies that it must be closer than a to ℓ. Module x applies a jumping rule, i.e., changes attachments from a to b in cases 2a and 2b, and changes attachments from b to a in cases 2c and 2d

```
Enter strip east
1
State=forward & Position(0,-1)=leader & Position(1,-1)=full & Position(1,0)=empty
Slide east & Strip_counter=1 & State=strip
```

```
Enter strip south-east
1
State=forward & Position(0,-1)=leader & Position(1,-1)=empty
Convex transition south east & Strip_counter=1 & State=strip
```

Fig. 19 Rules for starting the reconfiguration into canonical strip

4.2.4 Canonical strip rules

Once an active module reaches the leader, the reconfiguration rules do not apply any more since the DFS numbers cannot decrease further. New rules are needed to produce the canonical strip. The first two of these rules (refer to Fig. 19) change the state of the modules as they pass by the leader. The following two rules (refer to Fig. 20) make the strip grow. A final rule (refer to Fig. 21) is required to make the last module aware that the construction of the canonical strip has ended. For this purpose, the module needs to know in advance the total number of modules. This can either be autonomously computed by the modules as a preliminary task (Nichitiu et al. 2001; Wallner 2009) or on the fly through a simple modification of the rules `Enter strip`, by adding a postcondition in which the leader, which has this information (as pointed out in Sect. 4.1.4), transfers it to the modules when they advance past it. Notice that the priorities for this set of rules guarantee that rules `Enter strip` and `End strip` are always executed instead of `Advance strip`, if they simultaneously apply. At the end of this step, all modules form a strip. The state of all modules except the first (`leader`) and the last (`backward`) ones is `wait`.

The example in Fig. 22 shows the first steps of a synchronous version of the reconfiguration process. Step (i) illustrates the scan tree. From step (ii) on, the DFS number is represented. At step (iii), the leaves activate and the reconfiguration starts. Static modules and active modules are represented in different colors. The attachment of active modules to the static tree are depicted as thicker edges. Notice that the active module attached to the static module numbered 24, although active, cannot advance, since it sits on a branching static module, and the right hand rule advance would increase its potential function. In step (iv), a bottleneck situation can be seen: the active module attached to 31 is about to jump and attach to the static module labeled 14, as can be seen in step (v). A collision conflict happens between the active modules attached to 15 and 30 in step (viii): the first one has the preference. At step (ix) an active module gets stopped on top of the static branching module labeled 7, and it stays still until step (xvi), when the static branch in front of it disappears and it is free to advance again.

5 From canonical to goal

The reconfiguration from canonical strip to goal shape essentially reverses the procedure described in Sect. 4.2. Modules advance from the rightmost end of the strip following the left hand rule and the goal shape is constructed in a clockwise depth-first manner starting from the current leader, which will occupy the leader position in the final shape as well.

```
Advance strip east
2
State=strip & Position(0,-1)=full & Position(1,-1)=full & Position(1,0)=empty
Slide east & Advance Strip_counter
```

```
Advance strip south east
2
State=strip & Position(0,-1)=full & Position(1,-1)=empty
Convex transition south east & Advance Strip_counter & State=wait
```

Fig. 20 Rules for making modules advance along the canonical strip

```
End strip
1
State=strip & Position(0,-1)=full & Position(1,-1)=empty & strip_counter=last
Convex transition south east & State=backward
```

Fig. 21 Rule for ending the canonical strip

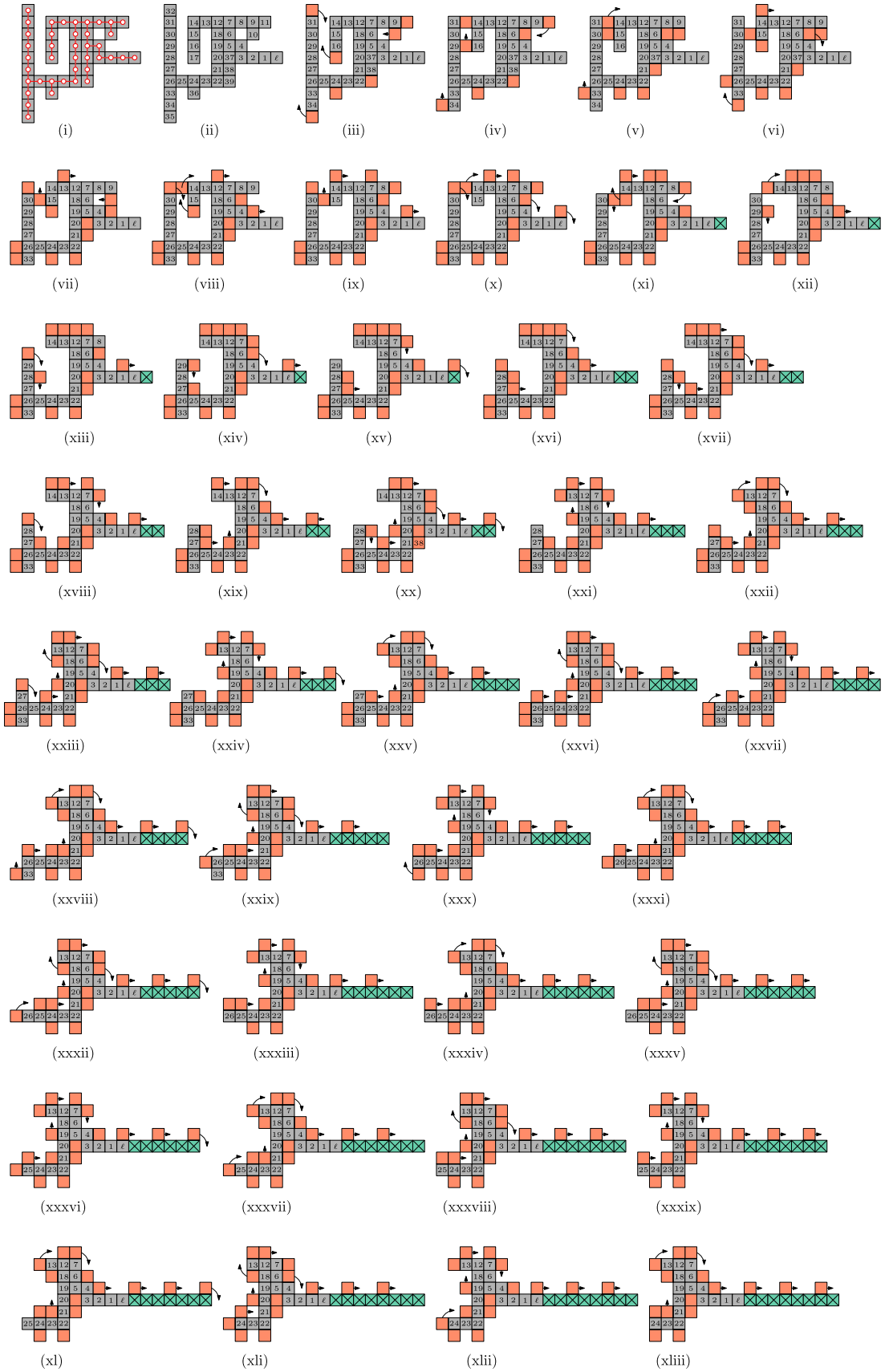


Fig. 22 First steps of a synchronous reconfiguration process (Color figure online)

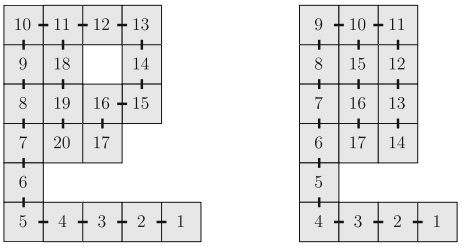


Fig. 23 Two examples of goal configurations. Recall that the number on a module indicates the order in which it arrives at that position. In the *left* configuration, the pairs of modules (8, 16) and (7, 17) form temporary bottlenecks, before modules 18, 19, and 20 reach their goal destination. In the *right* configuration, the pairs (8, 12), (7, 13) and (6, 14) form temporary bottlenecks before modules 15, 16 and 17 reach their goal destination

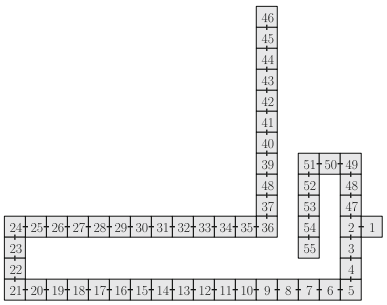


Fig. 24 When building this goal configuration, the active modules leading to positions 40–46 may encounter the active modules leading to positions 52–55 in the narrow passage, which is a bottleneck that has been created on the fly. The first modules are moving upwards, while the second ones are moving downwards, and this may cause a deadlock that needs to be handled

For this to be possible, at least one of the modules needs to know the goal shape. In the solution we propose, the leader has (receives or computes) the following information about the goal shape:

- The *clockwise* depth-first numbering of the scan tree of the goal configuration (as opposed to the counterclockwise numbering of the initial configuration in the forward step), the relative coordinates, and `Min` and `Max` values for each node.
- The position of the lbh-module of each hole. More precisely, in addition to the relative coordinates and numbering values mentioned above, a flag indicates whether a goal position corresponds to a lbh-module of a hole.
- The position of each pair of bottleneck modules. In other words, in addition to the information mentioned in the two previous items, one bit and two integers indicate (i) whether a goal position corresponds to the entrance/exit of a cul-de-sac and (ii) the relative position of the other goal position producing the bottleneck. Notice that in the reverse procedure, *temporary* bottlenecks may appear as the goal configuration is being formed. Such bottlenecks do not show up in the goal shape, but they are present at some point in an intermediate reconfiguration stage. The information about temporary bottlenecks needs to be known and relayed by the leader to the relevant modules. These bottlenecks can be easily detected when assigning DFS-values to the goal configuration modules. See Fig. 23.

The information itemized above is equivalent to computing the leader, the lbh-modules of holes, the scan tree, and the DFS numbering function, all described in Sect. 4.1, but now in a clockwise direction rather than counterclockwise.

When a module advances from the rightmost end of the strip following the left hand rule and moves past the leader, it receives from the leader its goal destination in relative coordinates as determined by the clockwise DFS numbering computed above (the rightmost module occupies the goal position with DFS number 1, the next module occupies DFS number 2, and so on). As the module advances, applying the reverse of the rules for forward reconfiguration, it updates its relative coordinates as it moves. This update allows the module to know when it has reached its final destination, at which point it becomes a static module.

During the reverse reconfiguration, the reverse rules do not need to take care of activation conflicts, as these conflicts do not have an analogous role in the reverse procedure; that is, deactivating a module when it gets to its final position does not generate a conflict. Collision conflicts may occur during the reverse procedure and they are handled analogously to the forward procedure: lower DFS number always has priority. However, obstruction conflicts will be handled differently in the forward and backward procedures, as will be seen below.

The main difference between the forward and the backward procedures is that in the case of the latter, modules advance to their goal destination in the order determined by the depth-first traversal of the final configuration (whereas in the forward procedure, the modules in the final strip are not necessarily ordered by their DFS numbers). As a result, we need to modify the jumping rules to make sure that no jump modifies the order of active modules as they advance along the boundary of the static modules. An additional issue that the backward procedure must address is that bottlenecks and their associated cul-de-sacs are created as the reconfiguration is taking place. Hence an active module that did not explicitly enter a cul-de-sac through a bottleneck may find itself inside a cul-de-sac (and will have to exit it) when a bottleneck subsequently forms because some module reaches its final position and becomes static. See Fig. 24.

One final issue with the backward procedure is that when a hole closes, thus breaking the boundary of the shape into different connected components, we need to ensure that no module gets trapped in the wrong connected component of

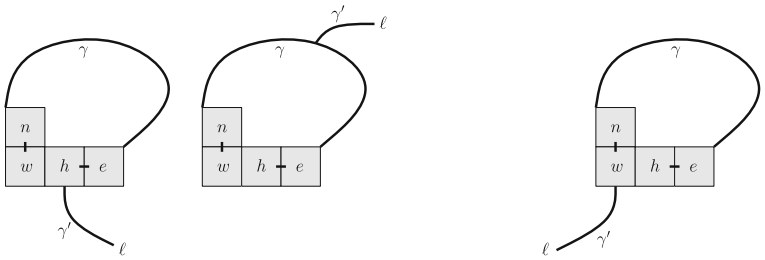


Fig. 25 The neighborhood of the lbh-module of a hole, the path γ connecting h and w in the scan tree, and the possible connections of γ with the leader through the path γ'

the boundary. We devote this section to describing all the differences between the forward and backward procedures, and will not describe again the issues that are common to the two.

5.1 Closing holes at the right time

We analyze here the configuration of the scan tree in the neighborhood of the lbh-module h of a hole. Since h is the leftmost among the bottommost modules along the boundary of the hole, it must necessarily have an east and a west neighbor, e and w respectively, and w must have a north neighbor n. This situation is illustrated in Fig. 25. In addition, in order to eliminate the cycle that the hole would produce, the scan tree guarantees that h and w are not directly connected. Due to the shape of the scan tree, it is easy to show that the path γ connecting h to w along the tree must necessarily go from h to e and from there to n and then to w, as illustrated in Fig. 25. Let γ' be the path in the scan tree connecting γ to the leader ℓ. If γ and γ' connect at a module other than w, it implies that h has a lower DFS number than w (Fig. 25 left). If they connect at w, then w has a lower DFS number than h (Fig. 25 right). Recall that the numbering is determined by a clockwise depth-first ordering of the scan tree.

In the first case, when $\texttt{Num}(h) < \texttt{Num}(w)$, the order of the modules filling the goal shape can be described as follows: First, all modules with DFS number up to $\texttt{Num}(h)$ in the goal shape are activated by the reverse reconfiguration algorithm. Note that all these nodes are external to the hole; that is, they lie in the left subtrees of the nodes along the path from the master ℓ up to module h. The subtree numbered 1 in Fig. 26(left) is an example of such a subtree. Then, if a subtree rooted at h exists, it is external to the hole [(as in the subtree numbered 2 in Fig. 25(left)], and the modules of this subtree are activated next. Note that all subsequent activated modules up to $\texttt{Num}(w)$ form the remainder of γ and all subtrees rooted at nodes of γ that lie within the hole, such as the subtree numbered 3 in Fig. 25(left). In other words, all modules internal to the cycle $\gamma \cup wh$ are placed next. Finally, w is placed, followed by all remaining modules external to the hole. Therefore, in this case it is important to make sure that w does not get to its position (thereby closing the hole) before all the modules in the subtree rooted at h have exited the hole. This can be easily controlled by a specific rule for modules h and w which only allows w to close the hole (i.e., to get to the position west of h) after h has detected that the module with DFS number equal to $\texttt{Max}(h)$ has advanced past it.

In the second case, when $\texttt{Num}(w) < \texttt{Num}(h)$, the modules filling the goal shape are sent in the following order. First, all modules with DFS number up to $\texttt{Num}(h)$ are activated, all of which are external to the hole. However, when h reaches its final position, the hole is closed, thus preventing modules in the right subtrees (if any) of the modules in γ from entering the hole and reaching their final position. This is illustrated in Fig. 26(right), where the modules in the subtrees numbered

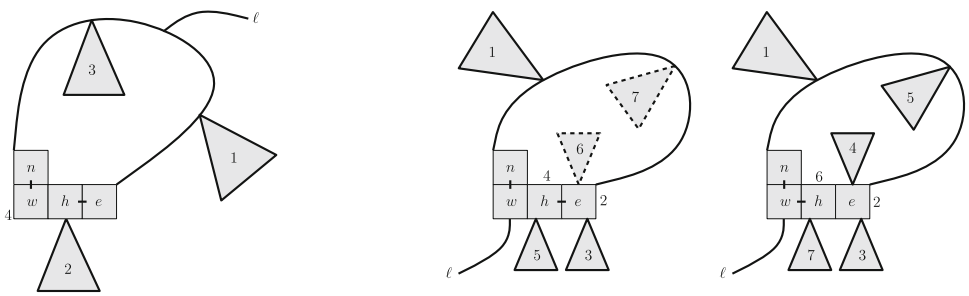


Fig. 26 *Left* The case where $\texttt{Num}(h) < \texttt{Num}(w)$. *Right* The case where $\texttt{Num}(w) < \texttt{Num}(h)$. In both cases, *triangles* indicate subtrees and *numbers* indicate the ordering of the modules in the reconfiguration. In the *left* case, all modules with final destination in the subtree labeled 2 need to advance past module h before module w is allowed to occupy its final position. In the *right case*, modules with final position in the *dashed* subtrees would never reach their destination. If h is connected to w and disconnected from e, then the *right* case becomes analogous to the *left* one and they do reach their destination

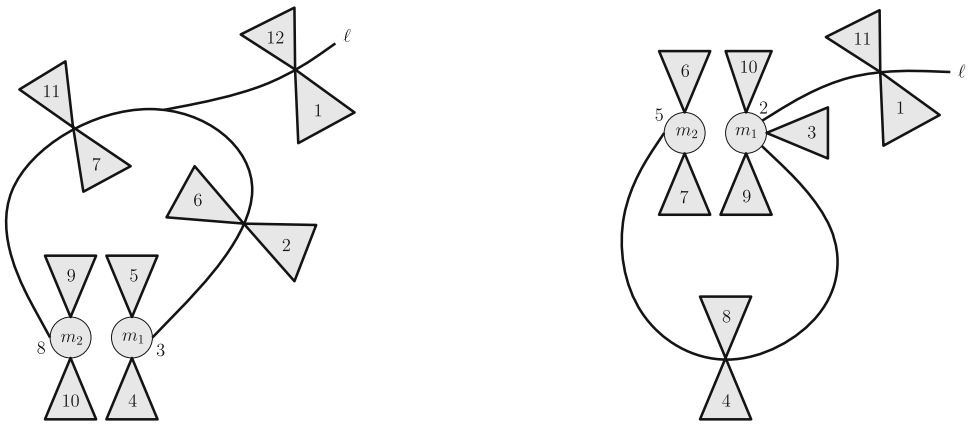


Fig. 27 *Left* When m_1 and m_2 have a least common ancestor distinct from m_1, no jumping is necessary. *Right* When m_1 is the least common ancestor of m_1 and m_2, all modules whose final destination is in subtree 6 wait for m_2 to reach its final position and then jump, while any other module advancing past m_1 follows its way without any jumping. In both cases, the modules m_1 and m_2 forming the bottleneck are symbolically represented by *circles*, *triangles* indicate subtrees, and *numbers* indicate the ordering of the modules in the reconfiguration

6 and 7 cannot enter the hole. The scan tree rules rectify this situation, which is detected whenever h is a lbh-module and $\texttt{Num}(w) < \texttt{Num}(h)$, by modifying the scan tree locally as follows: connect h to w and disconnect it from e, as illustrated in the rightmost scan tree in Fig. 26(right). The effect of this change is that the behavior of the algorithm in this case is now analogous to the previous case. Module h waits to occupy its final position (and hence close the hole) until e has detected that all modules in its left subtree (e.g., subtree 3 in the rightmost image in Fig. 26) have advanced past it and moved out of the hole. Again, the `Max` function can be used suitably to detect this situation. Observe that by the time h is ready to close the hole, all active modules whose final destination is within the hole (e.g., subtrees 4 and 5 in the rightmost image) are already in the hole.

5.2 Preventing deadlocks without changing the order of modules

In the forward reconfiguration, jump rules allow us to avoid deadlocks. The issue of deadlocks is relevant in the backward reconfiguration as well. However, since maintaining (DFS number) ordering of modules is crucial for the reverse reconfiguration algorithm, our solution to prevent deadlocks must ensure that it does not modify ordering.

Let m_1 and m_2 be two modules producing a bottleneck (recall Fig. 15) and let $\texttt{Num}(m_1) < \texttt{Num}(m_2)$. We distinguish two cases.

- If the least common ancestor of m_1 and m_2 in the scan tree is distinct from m_1, then observe that by the time m_2 reaches its lattice position and becomes static (thus forming a bottleneck with m_1), every module that wants to move into the cul-de-sac is a module whose final lattice position is within the cul-de-sac. See Fig. 27(left). Hence, no jump rule equivalent is needed in this case. However, it is possible that a module needs to exit the cul-de-sac (thus creating an obstruction conflict at the bottleneck) because it is advancing along the static modules within the cul-de-sac and "entered" it before the bottleneck was formed. For example, in Fig. 27(left), modules from subtree 4 could still be advancing through the cul-de-sac after m_2 has reached its final position. Hence, a module from subtree 4 could be exiting the cul-de-sac at the same time that a module from subtree 9 is entering it. In such a case, the exiting module (which has a lower DFS number) has priority.
  There is a situation in which a consecutive sequence of bottleneck pairs (a corridor, as in Fig. 24) could cause a potential deadlock. For example, several modules of subtree 9 could be entering a corridor while modules of subtree 4 are exiting. In this case, a pair of obstructing modules exchange ids as well as their local information (goal destination and other data) and their attachments. The effect of this rule is equivalent to each module advancing one step.
- If m_1 is the least common ancestor of m_1 and m_2 [(refer to Fig. 27(right)], all modules with DFS number greater than $\texttt{Num}(m_2)$ need special handling. This includes modules that lie outside the cul-de-sac (such as those in subtree 6) as well as those that lie within the cul-de-sac (such as those in subtrees 7, 8, and 9). We need to ensure that these modules do not produce deadlocks and that they maintain their order in the event of a jump movement. We avoid deadlocks by guaranteeing that no module that enters a cul-de-sac through the bottleneck defined by m_1 and m_2 needs to exit it. This requires that the modules that lie outside the cul-de-sac, such as those in subtree 6, will

now jump from m_1 to m_2 rather than moving into the cul-de-sac. When the module with DFS number $\texttt{Num}(m_2)+1$ attaches to m_1, it waits until m_2 has reached its lattice position and then makes the jump move. Observe that this maintains the order of all modules outside the cul-de-sac. At the same time, each module lying within the cul-de-sac, as in subtrees 7, 8, and 9, does not perform the jump move and advances within the cul-de-sac until it has reached its final position. A simple rule determines whether an active module attached to m_1 should wait and jump to m_2 or move into the cul-de-sac instead: if the DFS number of the module lies in the range $[\texttt{Num}(m_2)+1, \texttt{Max}(m_2)]$, it jumps to m_2; otherwise, it advances past m_1 into the cul-de-sac.

Observe that while this case could be handled in a manner similar to the previous case (allow a module to enter the cul-de-sac and handle conflicts between exiting and entering modules using DFS numbers), we choose to implement the wait-and-jump approach because it reduces the number of times modules have to exchange ids, and furthermore, avoids unnecessary walks around cul-de-sacs.

See Fig. 28 for an illustration of the reverse reconfiguration from canonical to goal shape. The output in the illustration was generated from our simulator, which is discussed further in Sect. 9 on experimental results.

6 Correctness

In this section we prove that the rules described in the previous sections allow reconfiguration between any pair of robotic systems with the same number of modules, with or without holes. We first state a result from (Nichitiu et al. 2001; Wallner 2009) pertaining to preliminary computation performed by the modules before they enter into the `to_strip` state and start the reconfiguration algorithm.

Proposition 1 *There exists a set of rules allowing any robotic system to detect the topmost module among its rightmost modules (Nichitiu et al. 2001; Wallner 2009).*

The previous rules can be adapted to also detect holes and their lbh-modules.

Proposition 2 *There exists a set of rules allowing any robotic system to detect the leftmost module among the bottommost modules of the boundary of each of its holes, if any.*

Proposition 3 *The rules proposed in Sect. 4.1.3 allow any robotic system to build a scan tree, a tree in which all leaves lie on the boundary of the configuration and at least one leaf lies on the external boundary.*

Proof The rules produce a graph in which all possible vertical logical connections between adjacent modules are present. If the configuration has no holes, each vertically connected component is horizontally connected to each of its neighboring vertical connected components through only one horizontal connection, namely the highest feasible one. Therefore, the resulting graph is connected and acyclic. If the configuration has holes, the fact that each lbh-module is not connected to its neighbor to the west guarantees acyclicity, while connectivity holds.

As for the location of the leaves of the scan tree, if a module does not lie on the boundary of the robotic configuration, then it necessarily has north and south neighbors. This implies that its degree in the scan tree is at least 2 and it cannot be a leaf. If the shape has no holes, all leaves belong to its external boundary. Otherwise, consider the lowest lbh-module. Either it is a leaf because it does not have a module to its south, or the lowest of the modules located vertically below it must be a leaf. In both cases, this leaf necessarily belongs to the external boundary of the configuration. □

Proposition 4 *The rules proposed in Sect. 4.1.4 number all the modules of the robotic system by their distance to the master in a counterclockwise depth-first traversal of the scan tree.*

Proof Follows immediately from the rules in Fig. 8, which increment the `Num` value at each module when it is encountered for the first time during the depth-first walk through the scan tree. □

Lemma 5 *At all times along the forward reconfiguration, the static tree, although pruned, stays a scan tree. In addition, the numbering of the modules along its external boundary increases counterclockwise from the leader.*

Proof Follows immediately from the structure of the scan tree and the fact that at any stage of the reconfiguration algorithm, the only nodes to be pruned from the scan tree are the newly activated nodes, which are always leaf nodes of the current static tree. The resulting tree thus remains a scan tree of the static modules. Since the nodes are numbered in rightmost first DFS order, it follows that the numbering of the static modules increases in counterclockwise order from the leader along the external boundary. □

A deadlock loop is a sequence of active modules $a_1, \ldots, a_k$ such that the reconfiguration rules would require a_i to occupy the lattice position of a_{i-1} (where $a_0 \equiv a_k$).

Lemma 6 *The forward reconfiguration rules cannot create deadlock loops.*

Proof In a deadlock loop $a_1, \ldots, a_k$, it is obviously impossible that $Num(a_i) > Num(a_{i-1})$ for all i, $1 \leq i \leq k$. Therefore, there exists an i such that $Num(a_i) < Num(a_{i-1})$. For

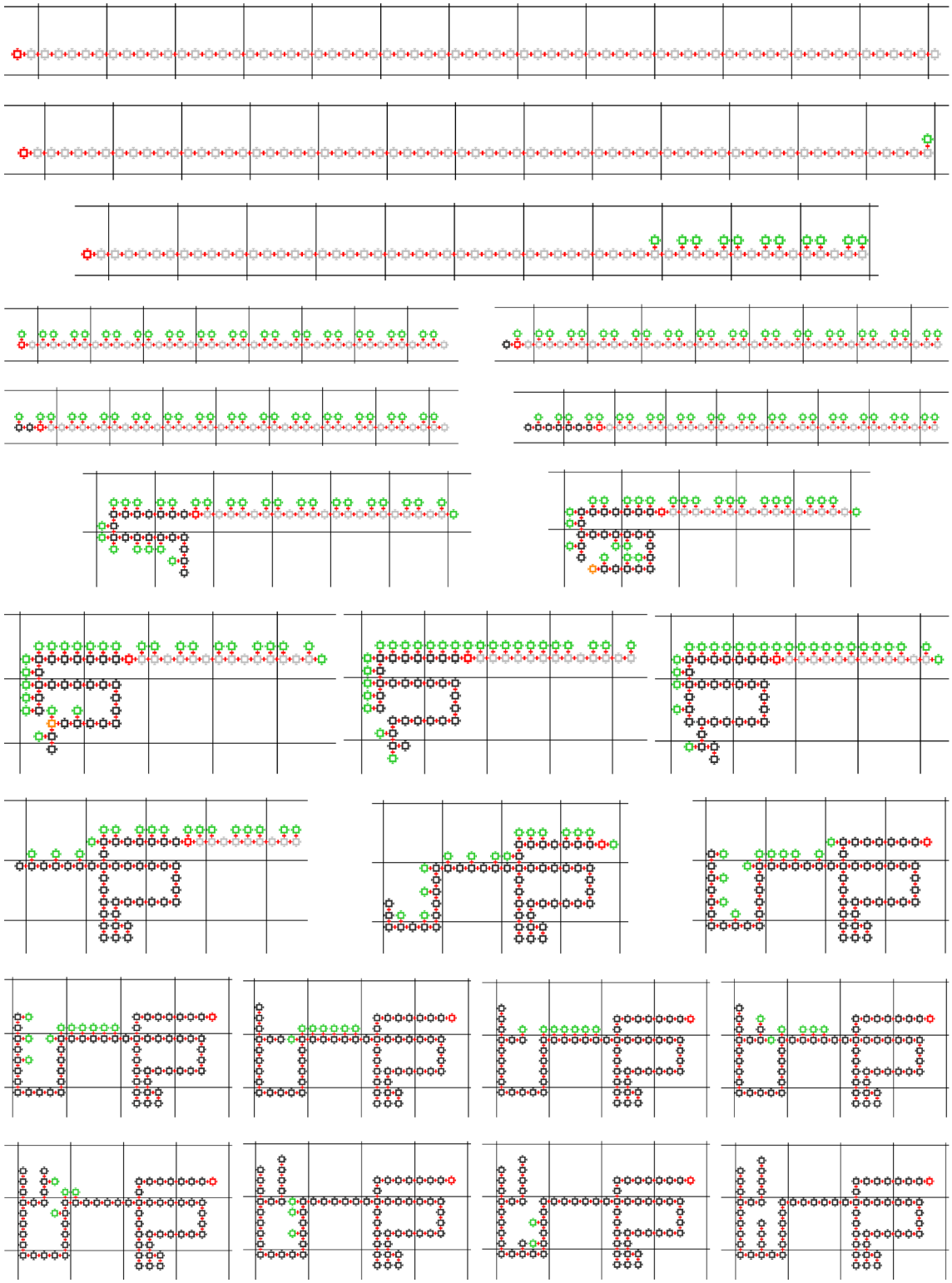


Fig. 28 Screenshots from our simulator illustrating a reconfiguration from the canonical strip (*top*) to the goal shape (*bottom right*). The three images in the *7th row* show how active modules wait before closing a hole, in order not to trap modules in it (see Sect. 5.1). The four images in the *9th row* show how active modules wait in front of a bottleneck which is about to be formed, and then jump over it (see Sect. 5.2). In the *last row*, the modules whose goal positions lie inside the cul-de-sac enter it (Color figure online)

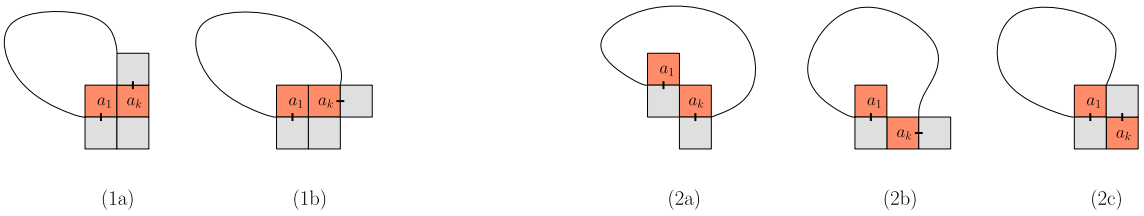


Fig. 29 *Left* two possible configurations for a loop in which a_1 intends to slide (1a and 1b). *Right* three possible configurations for a loop in which a_1 intends to perform convex transition (2a, 2b and 2c). Only the cases where a_1 is attached to its south are depicted. There exist analogous east, west, and north configurations (Color figure online)

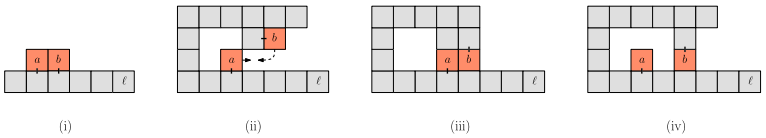


Fig. 30 If a is the first module in counterclockwise order from the leader ℓ, then (**i**) module b cannot activate in front of a; (**ii**) in a collision conflict with b, module a has priority and advances; (**iii**) if module b exchanges ids with a, the distance d decreases; (**iv**) if module b jumps in front of a, the distance d decreases (Color figure online)

the sake of clarity, let us assume that $Num(a_1) < Num(a_k)$. A simple case analysis, illustrated in Fig. 29, shows that the pair of modules, a_1 and a_k must be such that a_1 is exiting the cul-de-sac through the bottleneck, and a_k is entering it. However, the rules presented in Sect. 4 would prevent such situations, because jump rules ensure that module a_k does not enter the cul-de-sac. In particular, module a_k changes attachments in cases 1a, 2b, and 2c, whereas in cases 1b and 2a, module a_k will never be in the positions shown in Fig. 29. This is because those positions (i) cannot be ones at which it is activated, and (ii) cannot be reached by following the right hand rule with our advance and jump rules without a_k entering the cul-de-sac. It follows, therefore, that no deadlock loops can be produced. □

Proposition 7 *The forward reconfiguration rules to reconfigure the robot to the canonical shape cause all modules to advance past the leader.*

Proof Proposition 3 and Lemma 5 guarantee that there always exists a leaf of the static tree on the external boundary. As a consequence, there also exists at least one module that activates and travels along the external boundary. Let d be the distance (i.e., the number of edges) along the boundary between the leader ℓ and the first active module in the counterclockwise direction from ℓ. We will prove that d always decreases during the reconfiguration algorithm.

Among all active modules attached to the external boundary, let a be the first module in counterclockwise order from the leader ℓ. Notice that a is necessarily attached to the leftmost branch of the scan tree; i.e., the branch of the tree first traversed during the depth first traversal of the scan tree. For example, in Fig. 22, a would be the module with DFS number 10. The numbering decreases along the boundary from a to ℓ (see Lemma 5) and moreover, there cannot exist a leaf between a and ℓ. Therefore, there are no activations between a and ℓ and hence d cannot increase. If there are no conflicts or jumps, a can advance and d decreases by 1.

If a is stopped by a conflicting module b, we will see that b now becomes the first active module and hence d decreases. In fact, b cannot stop a in an activation conflict, since there exist no leaves between a and ℓ (see Fig. 30i). Similarly, b cannot stop a in a collision conflict since our choice of a guarantees that $Num(a) < Num(b)$ and hence a has priority (Fig. 30ii). As for obstructions, b cannot be advancing in front of a as that would imply that it is attached to the boundary between a and ℓ (Fig. 30i). If a and b are in a configuration that requires the exchange of ids, it follows that b now becomes the first active module and hence d decreases by 1 (Fig. 30iii). We are only left with the case that a and b are involved in a deadlock loop, which is impossible from Lemma 6. Finally, if module b applies a jump move, it must attach to the boundary between a and ℓ. In this case, b is the new first module and d decreases (Fig. 30iv). □

Proposition 8 *The rules proposed in Sect. 4 produce a horizontal strip to the right of the leader.*

Proof Follows immediately from Proposition 7. □

Lemma 9 *At any point during the reverse reconfiguration from the canonical strip to the goal configuration, the order of active modules along each connected component of the static boundary (counterclockwise if boundary is external and clockwise otherwise) is a subset of the initial numbering.*

Proof When no conflicts appear, the regular advance move cannot transpose the order since it only makes active modules advance one step forward along the sequence of edges of the boundary. Furthermore, waiting rules for closing holes and collision avoidance rules (recall that the module with lower DFS number has priority) also guarantee that numbering order is maintained. Finally, since jumping rules guarantee that when a module jumps a bottleneck during the reverse reconfiguration, no active module is attached to the boundary of the corresponding cul-de-sac, it follows that they do not produce any modification in the order of active modules. □

Proposition 10 *The rules proposed in Sect. 5 reconfigure the canonical horizontal strip into the goal shape.*

Proof Let $m_1, \ldots, m_n$ be the ordering of the modules along the strip, where m_n is the leader. We prove by induction on i that each module m_i reaches its final destination. The base case of $i = 1$ obviously holds because (i) Lemma 9 guarantees that m_1 advances past the leader and (ii) the final destination of m_1 is adjacent to m_n.

Assume by the inductive hypothesis that for all $i < k$, m_i eventually reaches its final destination. We prove that m_k reaches its destination as well. Consider any moment after m_{k-1} has reached its final destination. Let m_p be the parent of m_k in the goal scan tree. If m_k has not yet reached its final destination (contiguous to m_p), then it must be attached to the connected component of the boundary containing the edge e incident on m_p to which m_k needs to attach. The closing rule for holes guarantees that m_k cannot be isolated inside a hole boundary not containing e. Conversely, if e belongs in a hole, then that hole cannot close before m_k is attached to its boundary since the module that closes the hole must have DFS number greater than $\texttt{Num}(m_k)$. Hence, due to order invariance (see Lemma 9), it will close the hole only after m_k has entered it. Therefore, m_k belongs to the connected component of the boundary containing edge e. Notice that, because of Lemma 9, in the instance after m_{k-1} reaches its final destination, m_k must lie on the boundary as we go counterclockwise from ℓ to e. Let d be the distance (i.e., the number of edges) along the boundary between the current attachment of m_k and e. We will prove that distance d decreases.

Notice that d cannot increase, since that requires a branch of the goal tree to form between m_k and e. The modules in such a branch will have higher DFS number than k, and Lemma 9 guarantees that they cannot move ahead of m_k. If m_k can apply a rule, obviously d decreases (this includes the case of exchange of ids). We will prove that m_k can eventually apply a rule. A collision conflict with another module m_i could temporarily stop m_k, but only if $i < k$. By the inductive hypothesis, all modules m_i with $i < k$ reach their final destination. Therefore, collisions can only stop the advance of m_k temporarily. Another reason that may prevent m_k from advancing is if it is waiting to close a hole or is queueing behind a closing hole. In this case, the active modules m_i queueing in front of m_k all have $i < k$. By the inductive hypothesis, they all reach their final destination, and let m_k eventually move. Similarly, queueing in front of a bottleneck may prevent m_k from advancing. In this case, all active modules m_i that are entering the hole or queueing in front while m_k waits have $i < k$. By the inductive hypothesis, they all reach their final destination, allowing m_k to eventually move. The only remaining reason for m_k to wait would be deadlock loops, which cannot occur. In fact, in the strip-to-goal algorithm modules that enter a cul-de-sac only if their final destination is inside it. In other words, modules entering a cul-de-sac never exit it, making a deadlock loop impossible. It follows, therefore, that distance d always decreases. □

The following theorem follows immediately from Lemmas 8 and 10:

Theorem 11 *Given two robotic systems with the same number of modules, the rules described in Sects. 4 and 5 reconfigure one shape into the other.*

7 Complexity

There are several issues to be taken into account when analyzing the complexity of the proposed reconfiguration.

7.1 Memory and computation

The first phase of the reconfiguration (choosing a leader, detecting the holes and their lbh-modules, building the scan tree, and reconfiguring into a canonical strip) only requires $O(1)$ memory and computation for each module. The second phase (reconfiguring from the canonical strip to the final shape) requires $O(1)$ memory and computation for all modules except the leader, which needs either $O(n)$ memory (if it only needs to store the information of the goal shape) or $O(n)$ memory and computation (if it needs to compute the scan tree and DFS numbering of the goal shape on its own). Note that we assume that an integer fits in $O(1)$ space, which implies a bound on the number of modules of the robotic system. However, this is a natural limitation that cannot be avoided if the goal shape is not predefined.

7.2 Number of moves

Among the different actions each module needs to perform, physical moves such as slide or convex transition

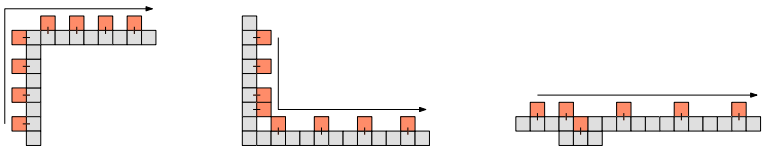


Fig. 31 The modules of the branch advance one after the other, separated by one (*left*), two (*center*), or three (*right*) empty spaces (Color figure online)

are the most energy and time consuming, when compared with attachments and detachments (such as in concave and opposite transitions), computing, or communication. It is hence important to realize that, in our reconfiguration, the number of physical moves each module performs is $O(n)$, where n is the total number of modules in the robotic system. In fact, the proof includes counting both position changes and attachment changes. Once activated, each module can attach to any of the static modules, which are at most $n-1$, and perform convex transition around it at most twice when proceeding towards the leader, but it can never return to the same static module, since the numbering of the substrate of an active module must decrease at each move. When forming the strip, the number of positions a module needs to attach to is always smaller than n. The same holds for the reverse reconfiguration. This bound is tight: reconfiguring a vertical strip into a horizontal one requires $\Omega(n)$ moves per module in any constant velocity and constant force solution (Aloupis et al. 2011).

7.3 Time steps

Consider a synchronized execution of the reconfiguration. Choosing a leader along the boundary can be done in $O(n)$ time steps (Wallner 2009). Consequently, detecting the holes and their lbh-modules can also be done in $O(n)$ time steps. The scan tree is built in $O(1)$ time steps, and the numbering ends in $O(n)$ time steps. We will prove that once the first leaves have been activated, the robotic system reconfigures into the canonical strip in a linear number of time steps. Reconfiguring the strip into the final shape requires the same number of time steps, and the entire reconfiguration runs in $O(n)$ time steps.

Let us start by considering the first active module in the counterclockwise direction from the leader (recall that it may not be the same module at all times). It is straightforward to prove that it passes by the leader in $O(n)$ steps. Since it can always move except when some other active module moves ahead of it (and thus becomes the new first module), it is clear that the first active module advances at least one position at each step. Since the number of possible positions for the first active module is linear (three different edges for each static module), it reaches the leader in $O(n)$ time steps.

In fact, if no active module had to wait during the reconfiguration, the first leaf of each tree branch (or the module successively replacing it by a jump) would reach the leader in $O(n)$ time steps for the same reasons discussed in the previous paragraph. Furthermore, all modules of that branch of the scan tree would advance one after the other, towards the leader, separated by 1, 2 or 3 empty spaces, depending on whether their advance consisted of only slide and convex transition moves, if they included single concave transitions, or they included consecutive concave transitions. The three possible cases are illustrated in Fig. 31. Hence, all modules of the robot would reach the leader in a linear number of steps.

In the case that there are waiting steps for some modules during the reconfiguration, we show below that after the first module reaches the leader, the last module of the robot will reach the leader in another $O(n)$ steps. Observe that the reason for a delay between two consecutive modules a and b arriving at the leader is one of the following:

- Module b is simply advancing behind the module a. In this case, there is only a constant delay between the arrival of a and b, as explained previously in Fig. 31.
- Module b has to wait because of a collision or obstruction conflict. We already know that this can happen only in bottlenecks. In this case, module b has to wait for the obstructing modules to move out and then either jump or, if the bottleneck has disappeared, enter the *cul-de-sac*. In the first case, there will only be constant space between b and its predecessor. In the second case, since the bottleneck has disappeared, it will not again cause a new delay to any other module. Furthermore, the delay caused to b due to this bottleneck is equal to the length of the cul-de-sac. It follows therefore that the overall sum of all such delays is proportional to the sum of the lengths of all cul-de-sacs, which is $O(n)$.

This proves that after $O(n)$ time steps all modules have reached the leader. Forming the strip is trivially done without obstructions, proving that the entire forward reconfiguration is done in $O(n)$ time steps.

7.4 Communication

Communication is a less costly activity, when compared with changing attachments and changing positions. Nevertheless, it is worth noticing that the communication performed by each module is linear, as a constant size communication is performed by the application of each rule on a module.

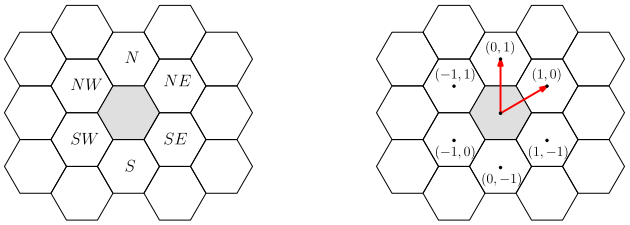


Fig. 32 The hexagonal setting. *Left* labels for the six neighbors of the *gray* module. *Right* a local reference system and relative positions of the neighbors of the gray module

8 Extension to the hexagonal setting

In this section, we give a brief overview of how our strategy adapts to the hexagonal case. Due to its similarity to the material presented in previous sections, we omit all proofs and concentrate on the issues that are specific to these lattices.

For the hexagonal case, each module of the robotic system is assumed to be centered at a point $(\sqrt{3}x/2, x/2 + y)$, where $x, y \in \mathbb{N}$. In this context, a grid position has 6 neighbors, which can be labeled for a more intuitive description of the procedures, but can also be designed using relative coordinates as required by the precise formulation of the rules. See Fig. 32 for an illustration.

Each module can be attached to or detached from any of its six neighbors, and the definitions of connectivity, holes, and hole/external boundary of a configuration can be easily adapted. Notice that, as opposed to the square case, two modules cannot be vertex-adjacent in a hexagonal lattice: they are either edge-adjacent or do not touch. As a consequence, hole boundaries and the external boundary are always pairwise disjoint. In fact, in the hexagonal case there is no such thing as a Moore neighborhood distinct from the von Neumann neighborhood of a module. This simplifies the moving rules, as there is no distinction between slide and convex transition: in the hexagonal case, the only possible move that modifies the position of a module consists of rotating around a static neighbor and proceeding from one neighboring position to a following one, either clockwise or counterclockwise (see Fig. 33a). As for changing attachments, the choice is wider than in the square case (see Fig. 33b).

In our model, changing position only requires the goal lattice position to be free (refer to Fig. 33c). Some current hexagonal prototypes of self-reconfiguring modular robots can perform this move with no extra empty space requirements. This is the case, for example, for the metamorphic robot (Pamecha et al. 1996). Some other current prototypes require some light extra empty space requirements, namely the cell adjacent to both the initial and the final positions should be free. This is the case, for example of Catoms (Kirbi et al. 2007) and, depending on the relative sizes of the components, of Fracta (Murata et al. 1994). Finally, some other current prototypes have the strong extra empty space requirement that in addition to the goal position, the following three lattice cells must also be empty: (i) the cell (say m) that is simultaneously adjacent to the intial and goal position, (ii) the cell that is adjacent to m and the initial position, and (iii) the cell that is adjacent to m and the goal position. Examples of these are HexBot (Sadjadi et al. 2009) and Fracta.

It is worth noticing that the classification we propose into these three categories (no/light/strong extra empty space requirements) applies to all current and potential prototypes of hexagonal-shaped robotic modules, and it is possible to classify every new design into one of the three. Therefore, it is interesting to see that any robotic system falling into one of these categories can perform our move without extra empty space requirements, as long as the units are grouped into meta-modules. For systems with no space constraints, meta-modules and units coincide. For systems with light constraints, meta-modules of just six units are enough (see Fig. 34, left). For systems with strong constraints, we propose meta-modules of eighteen units (see Fig. 34, right). Our meta-modules are inspired by Nguyen et al. (2000); to see how they perform the move, refer to the Appendix.

In addition, some chain-type modular robots such as M-Tran (Kurokawa et al. 2008), Polybot (Yim et al. 2000), and Superbot (Salemi et al. 2006) can arrange into hexagonal meta-modules with no extra empty space constraints. These meta-modules look like the ones in Fig. 34(left), where each of six hexagonal units are, in fact, spider-shaped: each *spider* is a three dimensional construction with a central "body" and

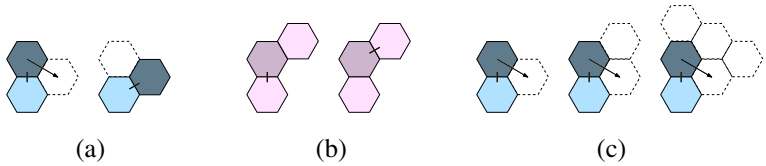


Fig. 33 The hexagonal moves. **a** Changing position to SE. Considering all orientations, there exist 5 analogous clockwise position changes, and 6 symmetric counterclockwise position changes. **b** Changing attachments from S to NE. There exist 4 more possible changes from S to the remaining neighbors, and 5 more possible orientations for the starting attachment. Considering all orientations and all initial/final attachments, there exist a total of 30 possibilities for changing attachments. **c** Space requirements for changing position. *Left* no extra empty space requirements. *Center* light extra empty space requirements. *Right* strong extra empty space requirements. In all cases, lattice cells required to be empty are marked by *dotted lines* (Color figure online)

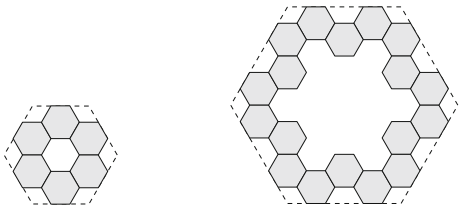


Fig. 34 *Left* when the units have light extra empty space requirements, meta-modules of six units are capable of changing positions without extra space requirements. *Right* when the units have strong extra empty space requirements, meta-modules of eighteen units are capable of changing positions without extra space requirements

six "legs" which can expand and contract. See Molina (2012) for details.

The pre-processing steps prior to the reconfiguration are straightforward. The leader of a connected configuration can be defined and found analogously to the square case, and so can the holes and their lbh-modules. As for the scan tree in the hexagonal case, all vertical connections are present, as in the square case. Vertical strips of modules connect to each other through the topmost possible connection. This means that each topmost module connects to its NE and NW neighbors (if they exist). The only exception to this rule is for lbh-modules: these do not connect to NW. The numbering is built analogously to the square case, by a depth-first traversal of the scan tree. This only requires adapting the rules to the fact that the nodes of the tree may have up to degree 6 instead of 4.

Adapting the reconfiguration strategy is easy. As the distinction between slide and convex transition disappears, the rules for changing position are simplified. Furthermore, collision conflicts are also simpler to deal with. It is easy to see that when a module intends to change position, only three other modules may collide with it (compare Fig. 35 with Figs. 13 and 14).

As for bottlenecks, the situation is similar to the square case, although vertex adjacency is impossible (see Fig. 36).

The canonical strip is built from the leader in the NE direction with analogous rules to the square ones. Finally, some of the correctness proofs in Sect. 6 are the same in the hexagonal case as in the square case, since they are of topological or combinatorial nature. This is the case, for example, for the proof of Lemma 5, which states the fact that the static tree, although pruned, stays a scan tree along the reconfiguration. It is also true of Proposition 7, which states that all modules pass by the leader when moving forwards. Some other proofs, which are based in a case analysis, require us

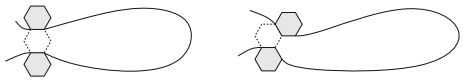


Fig. 36 Possible bottlenecks in the hexagonal setting

to consider more cases since a hexagonal module has six direct neighbors rather than four. However, some of the case analysis is simplified by the fact that no vertex adjacencies are possible. This is the case for Lemma 6, which states that deadlock loops cannot occur.

9 Experimental results

In this section we describe the main issues related to the implementation of our reconfiguration rules, and also present the results of some systematic experiments. The interested reader is encouraged to visit the companion web page to this paper at http://www-ma2.upc.edu/vera/local-reconfiguration/, where the following can be found:

1. The square and hexagonal simulators. Two simple and practical Java tools for simulating synchronized distributed algorithms on sets of 2-dimensional square/hexagonal lattice-based agents. The systems allow the user to define sets of modules and sets of rules and apply one to the other. The systems simulate the synchronized execution of the set of rules by all the modules, and can keep track of all actions made by the modules at each step, supporting consistency warnings and error checking. Both simulators come with user guides and examples. The source code is available, and can be modified as long as the following requirements are fulfilled:
   (a) The source code of the modified version of the simulator must be made public.
   (b) Acknowledgment to the original version and authors of the simulator must be included.
   (c) A notice must be sent by e-mail to the authors of the original simulator, explaining what the modification does and where it is available.
2. The sets of rules. The web page also provides implementations of the rules proposed in the previous sections, both for the square and the hexagonal case, together with a legend to facilitate understanding of the color codes of states, as well as the use of counters and message channels.

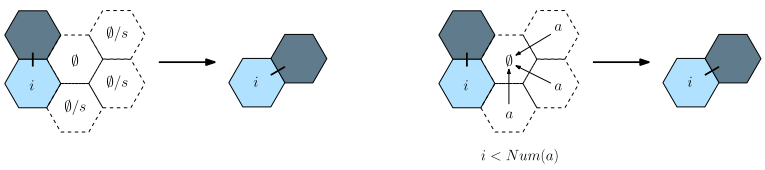


Fig. 35 Rules for changing position to SE (Color figure online)

3. A series of examples. These are initial and goal configurations designed to show most of the issues discussed in this paper, such as how to avoid activation conflicts and collisions, how to make a jump move at bottlenecks in front of cul-de-sacs, and so on.

The simulators offer a wide range of possibilities. In our implementation of the reconfiguring rules, though, only some of them are used. More precisely, each module can store and is able to modify the following information: a pair of integers, for its (relative) position (x, y); its connections to neighbors (attached/detached); its state (5 text characters); a number of counters (each being a 16-bit integer), which is 24 in the square case and 31 in the hexagonal case; two outgoing/incoming messages per neighbor (each a 16-bit integer), although 5 have been used for clarity. Each module can check whether a given grid position of its neighborhood is free or occupied by another module. In this last case, it can compare its own priority with the one of the neighboring module, check the state of the neighboring module, and compare the values stored in its own and its neighbor's counters. Finally, each module can execute the following instructions, which may be combined with *not*, *and*, and parenthesis: combine counters, incoming messages, and integers with $+$, $-$, $*$, /, max, min, and mod; compare counters, incoming messages, and integers with $\leq$, $<$, $\geq$, $>$ and $=$; test for matching of states and text, and of counters and integers. It is worth noticing that our simulation is synchronized. At each clock step, all modules check all rules and apply the chosen ones simultaneously. The simulation is also completely parallel.

When implementing the rules, some design decisions have been made in order to improve readability:

- *The use of steps* This is a simple way to deal with activation and collision conflicts. When a module intends to change position, it proceeds in two steps. First it applies a rule to declare its intentions, and then a rule to actually move from its current position to the new one, if possible (i.e., if the intended move produces no conflict or if the solution of the conflict allows the module to move). We consider this to be the simplest implementation of the procedure to check all neighbors' intentions before proceeding suitably.
- *The use of priorities* Using priorities is convenient, as priorities allow us to reduce the number of rules. In fact, we have used them as a way to simplify coding "else" options in the pre-conditions of the rules: a module that cannot apply a higher priority rule tries to apply rules of lower priority. Priorities are, therefore, not strictly necessary to produce the reconfiguration, although our implementation uses them. In the forwards reconfiguration, jumping has priority over advancing one step; taking care of activation and collision conflicts has priority over changing position; and holding messages has priority over all other possibilities.
- *Simultaneously applying several rules* When several rules with the same (highest) priority apply to a given situation, the module applies all of them. Again this option is not necessary, but it simplifies coding the rules because it reduces their number.
- *The use of colors* Our simulation uses colors to make the algorithm more visual. Colors are associated with states. This is just a visual artifact, but it means that we have implemented specific rules (with their priorities) to incorporate the use of colors.

It must be noted that all these decisions make the rules easier to read and the visualization of the process clearer, at the price of increasing the number of rules. Therefore, we do not make any claims about the optimality of the size of the rule sets.

In the previous sections, the reconfiguration strategy is described in stages. In practice, though, these stages overlap. For example, while a hole message is still being passed, some modules are already building the scan tree, and so on. When implementing the rules, this must be carefully taken care of. For example, the DFS numbering function cannot return from the leaves through a hole leader candidate if the latter has not yet been accepted or rejected; an active module cannot apply an advance rule if its neighboring static modules have not yet finished computing their numbering; and so on.

Finally, since our simulators do not support the existence of a module capable of storing a linear amount of information, we must simulate the fact that the leader needs enough information about the goal shape to carry out the reverse reconfiguration. In order to handle this issue, we have designed the following solution: (i) Place a translated copy of the goal shape to the left of the initial configuration so that their leader modules have the same y-coordinate, and find DFS numbers for the goal shape. (ii) Implement specific rules for the initial leader to locate the relative position of the goal leader (by horizontally scanning grid positions to its left until the goal leader is located). (iii) Each active module gets the value of this relative position from the initial leader. It can then use this value to determine if it has reached its goal destination by scanning the correct number of grid positions to the left to determine if a module with its DFS number exists at that position in the goal shape.

All together this gives rise to a set of 558 rules in the square case (1003 in the hexagonal case), only 198 (236) of which take care of the actual reconfiguration itself, while the rest are devoted to the preprocessing, as detailed in Table 1.

We have run a systematic set of simulations on several shapes, namely vertical lines, horizontal lines and the configurations illustrated in Fig. 37. The vertical line and the

Table 1 Number of implemented rules

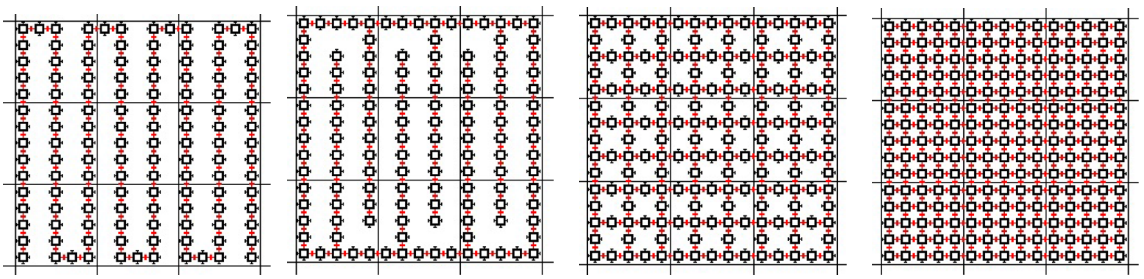
Stage	Rules (square grid)	Rules (hexagonal grid)
Elect leader, detect holes, form scan tree (both initial and goal shapes)	129	216
DF-number the initial shape	105	256
DF-number the goal shape	126	296
Reconfigure forwards (initial shape to strip)	109	166
Reconfigure backwards (strip to goal shape)	89	70

horizontal line respectively are one column and one row of the dense square. For each one of the shapes, we have run the algorithm for $n = 10, 20, 50, 100, 200, 500$, and 1000, where n is the total number of modules. In all cases, the initial shape and the goal shape have been the same. In other words, we have converted the shape into a horizontal strip and then we have reconstructed it again. Table 2 shows the number of parallel steps needed in our simulation. This includes both the preprocessing steps (elect a leader, detect holes, and elect a lbh-module for each hole, build the scan tree, and get each module to know its numbering values), as well as the actual reconfiguration steps (from initial shape to canonical, from canonical to goal shape). Figure 38 shows the graph of the results in Table 2. We do not show detailed results for the hexagonal case, because they are analogous. For some shapes, depending on their orientation, the number of steps for the square and hexagonal versions of the same shape is almost identical, and in some others it is scaled by a factor between 1.2 and 1.3 due to the increased number of rotations of active modules around one of the static ones.

10 Conclusions, discussion, and open problems

We have proved that, within a square or hexagonal lattice, it is possible to reconfigure any connected set of modules into any other connected set containing the same number of modules by means of a distributed algorithm based on local rules. Furthermore, the number of moves, communication, and time steps required is linear in the number of modules.

Notice that our discussion and our implementation of the algorithm are synchronous. Nevertheless, obtaining an asynchronous implementation is quite straightforward. In the first place, all counting rules are based on message passing, as opposed to counting clock steps. Therefore, the only issues to be taken care of in order to desynchronize the algorithm are activation and collision conflicts, and this can be done by means of shaking hands procedures, at the price of increasing communication.



(a) Snake (b) Square: snake hole (c) Square: little holes (d) Square: dense

Fig. 37 Configurations for the experiments

Table 2 Number of parallel steps for the entire reconfiguration (including preprocessing steps)

n	Horizontal line	Vertical line	Snake	Square: snake hole	Square: little holes	Square: dense
10	166	172	170	159	165	162
20	346	372	340	333	314	320
50	886	972	920	832	792	804
100	1.786	1.972	1.843	1.684	1.585	1.584
200	3.586	3.972	3.699	3.328	3.139	3.125
500	8.986	9.972	9.357	8.392	7.722	7.711
1.000	17.986	19.972	18.750	16.978	15.338	15.322

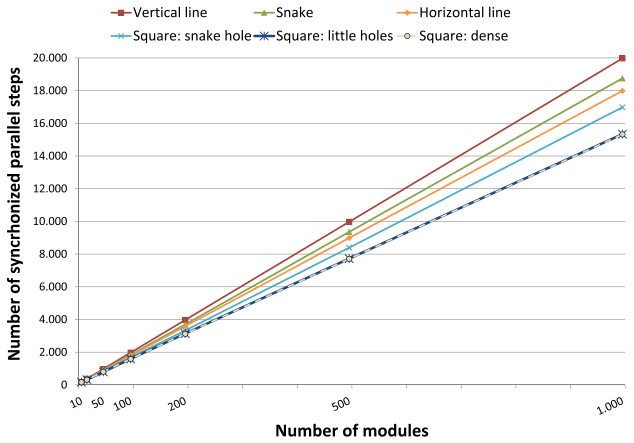


Fig. 38 Number of parallel steps used for the entire reconfiguration (including preprocessing steps)

Among the different actions a module needs to take, communication probably is the least time consuming one, when compared to changing attachments or changing position. Nevertheless, it is still an issue to be discussed. In our description of the algorithm as well as in our implementation, we assume that communication is possible between a module and any of the modules lying in its lattice neighborhood of radius two. This is an option that we adopt to simplify the description of the rules. In fact, it could be implemented in several different ways. One option could be as follows: once the leader has been chosen, all modules get to know their relative positions with respect to the leader by message passing. In fact this can be simultaneously done when electing the leader (Nichitiu et al. 2001; Wallner 2009). This information could then be used as ids, allowing modules to communicate to each other by bounded reach broadcasting. Another possibility is to use the DFS numbering as ids, once the leader has been elected. Other alternative solutions can be envisaged, but we do not intend to exhaustively explore them here, only to point out that this is a solvable issue.

We wish to highlight two open problems. One is related to the fact that our strategy uses a canonical intermediate shape. We propose it to be a strip, but it could be any other pre-defined simple shape, such as a rectangle. However, any choice of canonical shape implies that our strategy is not "in-place". The problem of designing an in-place universal reconfiguration algorithm using local rules is of great interest, both theoretically and practically, since the availability of additional space required by an algorithm such as ours may not always be guaranteed in certain physical environments. One of the challenging issues for an in-place reconfiguration algorithm is that it is more difficult to determine the final position for each module via local rules while also avoiding obstructions and deadlocks. An intriguing possibility is that if we allow the "tunneling" of modules, the approach presented in Aloupis et al. (2011) might be adaptible. An alternative strategy might be to develop a reconfiguration algorithm that only uses some constant size (preferably equal to 1) buffer around the initial and goal shapes. The other open problem is to extend this work to three dimensions, where the right hand rule does not apply in a straightforward way. A successful strategy in three dimensions would require a new approach. The reconfiguration algorithm presented in Aloupis et al. (2011) works in both two and three dimensions, but it strongly relies on the "tunneling" capability.

Acknowledgments The authors wish to explicitly thank the students Reinhard Wallner, Óscar Rodríguez, Sergio Ordóñez and Ángel Rodríguez for their precise work in implementing simulators and simulations. We also wish to thank an anonymous referee for detailed comments that helped to improve the readability of the paper. Suneeta Ramaswami was partially supported by NSF grant CCF-0830589. Ferran Hurtado and Vera Sacristán were partially supported by Projects MTM2012-30951, Gen. Cat. DGR 2009SGR1040, and ESF EUROCORES programme EuroGIGA, CRP ComPoSe: MICINN Project EUI-EURC-2011-4306, for Spain.

Appendix

10.1 Moves for the 8 sliding square units meta-module

Figure 39 shows how a meta-module made of 8 sliding square units can perform the slide move. Figure 40 shows

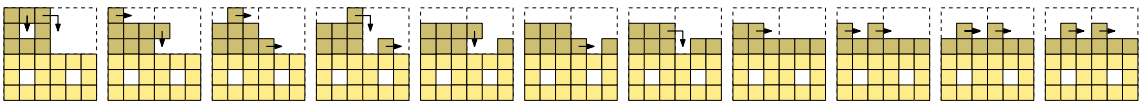


Fig. 39 A meta-module of 8 sliding square units performs *slide*. Only the first half steps are shown, the rest of the move is completed by symmetry (Color figure online)

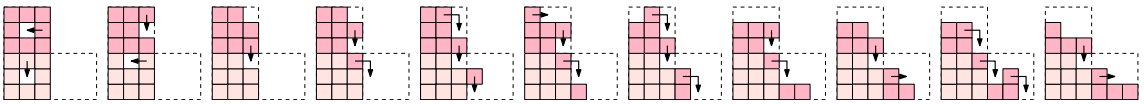


Fig. 40 A meta-module of 8 sliding square units performs *convex transition* without extra empty space requirements. Only the first half steps are shown, the rest of the move is completed by symmetry (Color figure online)

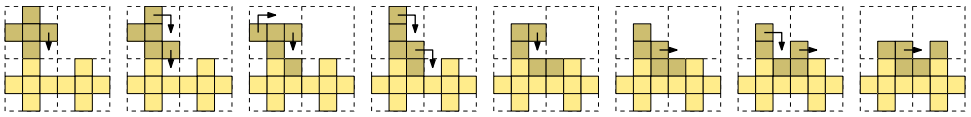


Fig. 41 A meta-module of 5 sliding square units performs *slide*. Only the first half steps are shown, the rest of the move is completed by symmetry (Color figure online)

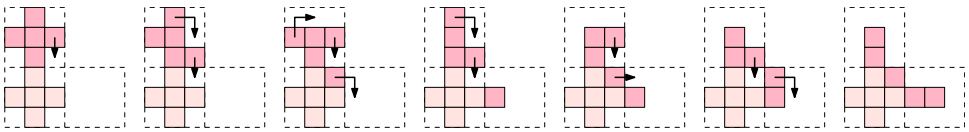


Fig. 42 A meta-module of 5 sliding square units performs *convex transition* without without extra empty space requirements. Only the first half steps are shown, the rest of the move is completed by symmetry (Color figure online)

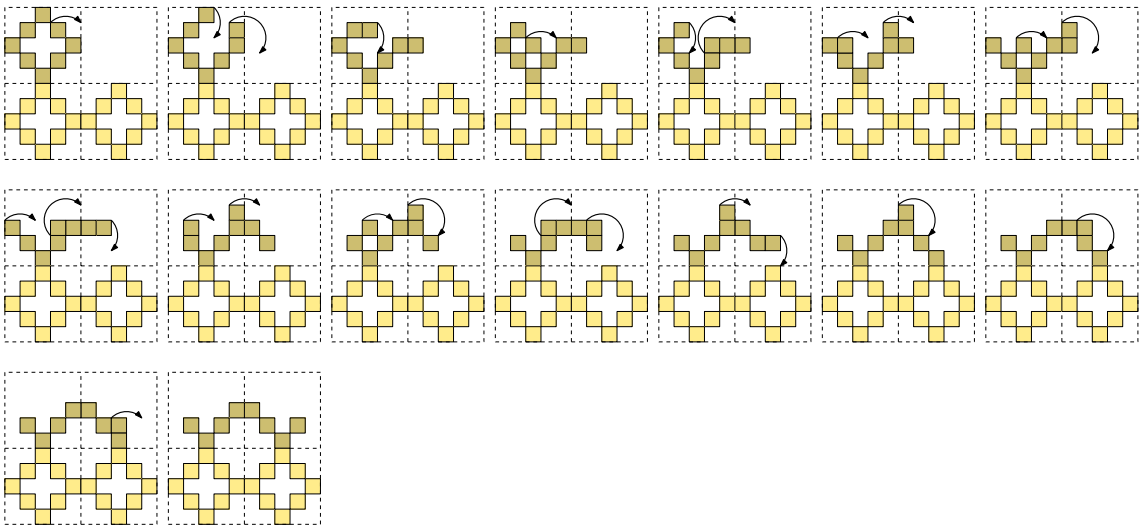


Fig. 43 A meta-module of 8 rotating square units performs *slide* without extra empty space requirements. Only the first half steps are shown, the rest of the move is completed by symmetry (Color figure online)

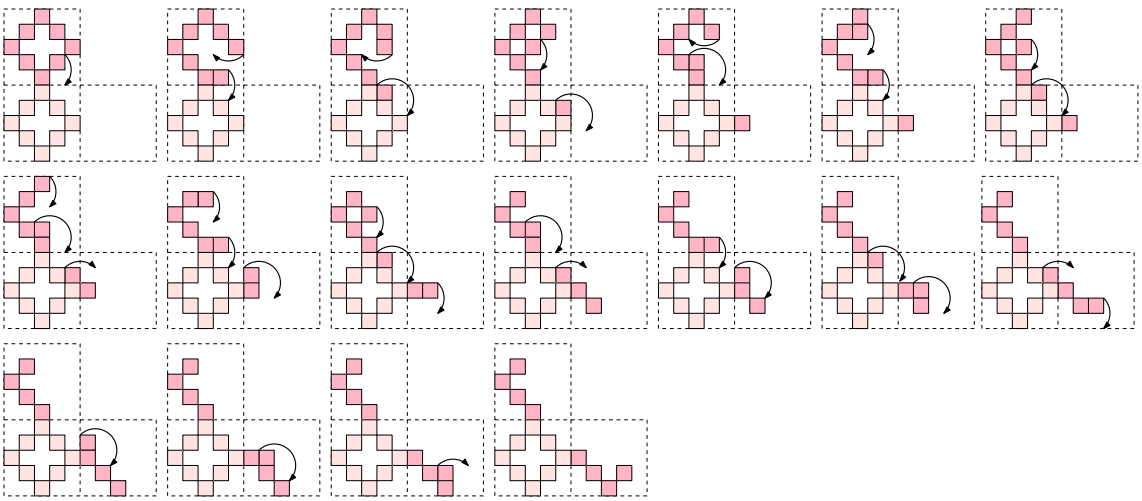


Fig. 44 A meta-module of 8 rotating square units performs *convex transition* without extra empty space requirements. Only the first half steps are shown, the rest of the move is completed by symmetry (Color figure online)

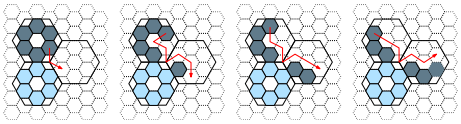


Fig. 45 When the hexagonal units have *light* extra empty space requirements, meta-modules of six units are capable to *change position* without extra empty space requirements. Only the first half of the atomic moves are shown, the remaining are obtained by symmetry (Color figure online)

how the same meta-module can perform the convex transition move without any extra space requirement.

10.2 Moves for the 5 sliding square units meta-module

Figure 41 shows how a meta-module made of 8 sliding square units can perform the slide move. Figure 42 shows how the same meta-module can perform the convex transition move without any extra empty space requirement.

10.3 Moves for the rotating square units meta-module

Figure 43 shows how a meta-module made of 8 rotating square units can perform the slide move without any extra empty space requirement. Figure 44 shows how the same meta-module can perform the convex transition move without any extra empty space requirement.

10.4 Moving meta-modules of hexagonal units

Figure 45 shows how a meta-module made of 6 hexagonal units can *change position* without any extra empty space requirement, when the units have light extra empty space requirements. Figure 46 shows how a meta-module made of 18 hexagonal units can *change position* without any extra empty space requirement, when the units have strong extra empty space requirements.

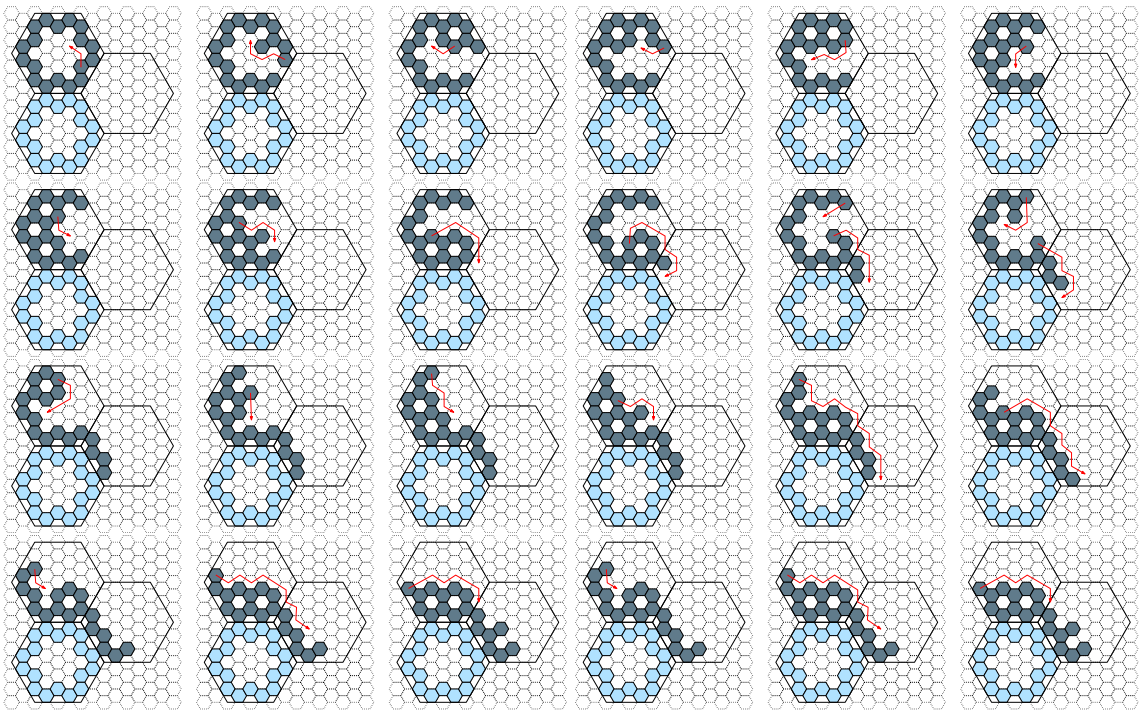


Fig. 46 When the hexagonal units have strong extra empty space requirements, meta-modules of eighteen units are capable to *change position* without extra empty space requirements. Only the first half of the atomic moves are shown, the remaining are obtained by symmetry (Color figure online)

References

Abel, Z., & Kominers, S. D. (2008). Pushing hypercubes around. CoRR abs/0802.3414.

Aloupis, G., Benbernou, N., Damian, M., Demaine, E., Flatland, R., Iacono, J., et al. (2013). Efficient reconfiguration of lattice-based modular robots. *Computational Geometry—Theory and Applications*, *46*(8), 917–928.

Aloupis, G., Collette, S., Damian, M., Demaine, E. D., Flatland, R., Langerman, S., et al. (2011). Efficient constant-velocity reconfiguration of crystalline robots. *Robotica*, *29*(1), 59–71.

An, B. K. (2008). Em-cube: Cube-shaped, self-reconfigurable robots sliding on structure surfaces. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3149–3155.

Bateau, J., Clark, A., McEachern, K., Schutze, E., & Walter, J. (2012). Increasing the efficiency of distributed goal-filling algorithms for self-reconfigurable hexagonal metamorphic robots. In *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications*, pp. 509–515.

Benbernou, N. M. (2011). *Geometric algorithms for reconfigurable structures*. Ph.D. thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge (MA), USA.

Beni, G. (1988). The concept of cellular robotic system. In *Proceedings of the IEEE International Symposium on Intelligent Control*, pp. 57–62.

Bhat, P. S., Kuffner, J., Goldstein, S. C., & Srinivasa, S. S. (2006). Hierarchical motion planning for self-reconfigurable modular robots. In *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 886–891.

Bishop, J., Burden, S., Klavins, E., Kreisberg, R., Malone, W., Napp, N., & Nguyen, T. (2005). Programmable parts: A demonstration of the grammatical approach to self-organization. In *Proceedings of the IEEE/RSJ International conference on intelligent robots and systems (IROS)*, pp. 3684–3691.

Bojinov, H., Casal, A., & Hogg, T. (2000). Emergent structures in modular self-reconfigurable robots. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, pp. 1734–1741.

Butler, Z., Fitch, R., & Rus, D. (2002). Distributed control for unit-compressible robots: Goal-recognition, locomotion and splitting. *IEEE/ASME Transaction on Mechatronics*, *7*(4), 418–430.

Butler, Z., Kotay, K., Rus, D., & Tomita, K. (2004). Generic decentralized control for lattice-based self-reconfigurable robots. *The International Journal of Robotics Research*, *23*, 919–937.

Butler, Z., & Rus, D. (2003). Distributed planning and control for modular robots with unit-compressible modules. *The International Journal of Robotics Research*, *22*(9), 699–715.

Castano, A., Shen, W.-M., & Will, P. (2000). CONRO: Towards deployable robots with inter-robots metamorphic capabilities. *Autonomous Robots*, *8*(3), 309–324.

Chiang, C. J., & Chirikjian, G. (2001). Modular robot motion planning using similarity metrics. *Autonomous Robots*, *10*, 91–106.

Dewey, D. J., Ashley-Rollman, M. P., De Rosa, M., Goldstein, S. C., Mowry, T. C., Srinivasa, S. S., Pillai, P., & Campbell, J. (2008). Generalizing metamodules to simplify planning in modular robotic systems. In *Proceedings of the IEEE/RSJ internacional conference on intelligent robots and systems (IROS)*, pp. 1338–1345.

Dumitrescu, A., & Pach, J. (2004). Pushing squares around. In *Proceedings of the 20th annual ACM symposium on computational geometry (SoCG)*, pp. 116–123.

Dumitrescu, A., Suzuki, I., & Yamashita, M. (2004). Formations for fast locomotion of metamorphic robotic systems. *The International Journal of Robotics Research*, *23*(6), 583–593.

Dumitrescu, A., Suzuki, I., & Yamashita, M. (2004). Motion planning for metamorphic systems: Feasibiity, decidability, and distributed reconfiguration. *IEEE Transactions on Robotics and Automation*, *20*(3), 409–418.

Fitch, R., & Butler, Z. (2008). Million module march: Scalable locomotion for large self-reconfiguring robots. *The International Journal of Robotics Research*, *27*(3–4), 331–343.

Fukuda, T., Ueyama, T., Kawauchi, Y., & Arai, F. (1992). Concept of cellular robotic system (CEBOT) and basic strategies for its realization. *Computers and Electronical Engineering*, *18*(1), 11–39.

Gilpin, K., Kotay, K., Rus, D., & Vasilescu, I. (2008). Miche: Modular shape formation by self-disassembly. *The International Journal of Robotics Research*, *27*(3–4), 345–372.

Hosokawa, K., Tsujimori, T., Fujii, T., Kaetsu, H., Asama, H., Kuroda, Y., & Endo, I. (1998). Self-organizing collective robots with morphogenesys in a vertical plane. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, pp. 2858–2863.

Ivanov, P., & Walter, J. (2010). Layering algorithm for collision-free traversal using hexagonal self-reconfigurable metamorphic robots. In *Proceedings of IEEE/RSJ international conference on robots and systems (IROS)*, pp. 521–528.

Jorgensen, M. W., Esben, E. H., Ostergaard, H., & Lund, H. H. (2004). Modular ATRON: Modules for a self-reconfigurable robot. In *Proceedings of IEEE/RSJ international conference on robots and systems (IROS)*, pp. 2068–2073.

Kirbi, B. T., Aksak, B., Goldstein, S. C., Hoburg, J. F., Mowry, T. C., & Pillai, P. (2007). Modular robots using magnetic force effects. In *Proceedings of the IEEE/RSJ international conference on intellingent robots and sytems (IROS)*, *3*, 2787–2793.

Kotay, K., & Rus, D. (2004). Generic distributed assembly and repair algorithms for self-reconfiguring robots. In *Proceedings of the IEEE international conference on intelligent robots and systems (IROS)*, *3*, 2362–2369.

Kotay, K. D., & Rus, D. L. (2000). Algorithms for self-reconfiguring molecule motion planning. In *Proceedings of the IEEE/RSJ international conference on inteligent robots and systems (IROS)*, pp. 2184–2193.

Kurokawa, H., Tomita, K., Kamimura, A., Kokaji, S., Hasuo, T., & Murata, S. (2008). Distributed self-reconfiguration of M-TRAN III modular robotic system. *The International Journal of Robotics Research*, *27*, 373–386.

Liu, J., Ma, S., Lu, Z., Wang, Y., Li, B., & Wang, J. (2005). Design and experiment of a novel link-type shape shifting modular robot series. In *Proceedings of the IEEE international conference on robotics and biomimetics (ROBIO)*, pp. 318–323.

Molina, E. (2012). *Self-reconfiguration of hexagonal lattice-based modular robotic systems*. Master's thesis, Facultat de Matemàtiques i Estadística, Universitat Politècnica de Catalunya, Barcelona, Spain.

Mondada, F., Pettinaro, G. C., Guignard, A., Kwee, I., Floreano, D., Deneubourg, J.-L., et al. (2004). Swarm-bot: A new distributed robotic concept. *Autonomous Robots, special Issue on Swarm Robotics*, *17*(2–3), 193–221.

Murata, S., & Kurokawa, H. (2012). *Self-organizing robots, volume 77 of Springer tracts in advanced robotics*. Berlin: Springer.

Murata, S., Kurokawa, H., & Kokaji, S. (1994). Self-assembling machine. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, pp. 442–448.

Murata, S., Kurokawa, H., Yoshida, E., Tomita, K., & Kokaji, S. (1998). A 3-D self-reconfigurable structure. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, pp. 432–439.

Murata, S., Yoshida, E., Kurokawa, H., Tomita, K., & Kokaji, S. (2001). Self-repairing mechanical systems. *Autonomous Robots*, *10*, 7–21.

Nguyen, A., Guibas, L. J., & Yim, M. (2000). Controlled module density helps reconfiguration planning. In *Proceedings of the international workshop on algorithmic foundations of robotics (WAFR)*, pp. 23–36.

Nichitiu, C., Mazoyer, J., & Rémila, E. (2001). Algorithms for leader election by cellular automata. *Journal of Algorithms*, *41*(2), 302–329.

Ordóñez, S. (2013). *Simulación de algoritmos distribuidos para sistemas robóticos modulares en retículos hexagonales*. (*Simulating distributed algorithms for modular robotic systems on hexagonal lattices*) Degree thesis, Facultat de Matemàtiques i Estadística, Universitat Politècnica de Catalunya, Barcelona, Spain.

Pamecha, A., Chiang, C., Stein, D., & Chirikjian, G. (1996). Design and implementation of metamorphic robots. In *Proceedings of the ASME design engineering technical conference and computers in engineering conference*.

Rodríguez, O. (2013). *Simulació de l'actuació distribuïda de robots modulars. (Simulating distributed actuation of modular robots)* Degree thesis, Facultat d'Informàtica de Barcelona, Universitat Politècnica de Catalunya, Barcelona, Spain.

Rus, D., & Vona, M. (2001). Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots*, *10*(1), 107–124.

Sadjadi, H., Mohareri, O., Amin Al-Jarrah, M., & Assaleh, K. (2009). Design and implementation of HexBot: A modular self-reconfigurable robotic system. In *Proceedings of the sixth international symposium on mechatronics and its applications*, pp. 1–6.

Salemi, B., Moll, M., & Shen, W.-M. (2006). Superbot: A deployable, multi-functional, and modular self-reconfigurable robotic system. In *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 3637–3641.

Sproewitz, A., Billard, A., Dillenbourg, P., & Ijspeert, A. J. (2009). Roombots-mechanical design of self-reconfiguring modular robots for adaptive furniture. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, pp. 4259–4264.

Støy, K. (2006). How to construct dense objets with self-reconfigurable robots. In *Proceedings of the European robotics symposium*, pp. 27–37.

Støy, K. (2006). Using cellular automata and gradients to control self-reconfiguration. *Robotics and Autonomous Systems*, *54*, 135–141.

Suh, J. W., Homans, S. B., & Yim, M. (2002). Telecubes: Mechanical design of a module for self-reconfigurable robotics. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, pp. 4095–4101.

Terada, Y., & Murata, S. (2008). Automatic modular assembly system and its distributed control. *The International Journal of Robotics Research*, *27*(3–4), 445–462.

Tomita, K., Murata, S., Kurokawa, H., Yoshida, E., & Kokaji, S. (1999). Self-assembly and self-repair method for a distributed mechanical system. *IEEE Transactions on Robotics and Automation*, *15*(6), 1035–1045.

Ünsal, C., Kiliççöte, H., & Khosla, P. K. (1999). I(ces)-cubes: A modular self-reconfigurable bipartite robotic system. In *Proceedings of SPIE, sensor fusion and decentralized control in robotic systems II*, pp. 258–269.

Vassilvitskii, S., Yim, M., & Suh, J. (2002). A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, pp. 117–122.

Wallner, R. (2009). *A system of autonomously self-reconfigurable agents*. Degree thesis, Institute for Software Technology, Graz University of Technology, Graz, Austria.

Walter, J. E., Tsai, E. M., & Amato, N. M. (2005). Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots. *IEEE Transactions on Robotics*, *21*(4), 621–631.

Walter, J. E., Welch, J. L., & Amato, N. M. (2004). Distributed reconfiguration of metamorphic robot chains. *Distributed Computing*, *17*, 171–189.

White, P. J., Kopanski, K., & Lipson, H. (2004). Stochastic self-reconfigurable cellular robotics. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, *3*, 2888–2893.

Yim, M., Duff, D. G., & Roufas, K. D. (2000). Polybot: a modular reconfigurable robot. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, pp. 514–520.

Yim, M., Lamping, J., Mao, E., & Chase, J. G. (1997). Rhombic dodecahedron shape for self-assembling robots. Technical report, Xerox PARC SPL TechReport P9710777.

Yim, M., Shen, W.-M., Salemi, B., Rus, D., Moll, M., Lipson, H., et al. (2007). Modular self-reconfigurable robots systems: Challenges and opportunities for the future. *IEEE Robotics and Automation Magazine*, *14*(1), 43–52.

Yim, M., Zhang, Y., Lamping, J., & Mao, E. (2001). Distributed control for 3D shape metamorphosis. *Autonomous Robots*, *10*(1), 41–56.

Yoshida, E., Kokaji, S., Murata, S., Tomita, K., & Kurokawa, H. (2000). Miniaturization of self-reconfigurable robotic system using shape memory alloy actuators. *Journal of Robotics and Mechatronics*, *12*(2), 96–102.

Zhang, H., González-Gómez, J., Xie, Z., Cheng, S., & Zhang, J. (2008). Development of a low-cost flexible modular robot GZ-I. In *Proceedings of the IEEE/ASME international conference on advanced intelligent mechatronics*, pp. 223–228.

Zykov, V., Chan, A., & Lipson, H. (2007). Molecubes: An open-source modular robotics kit. In *Proceedings of IROS-2007 self-reconfigurable robotics workshop*.

Ferran Hurtado (1951–2014) was a full professor in Applied Mathematics at the Universitat Politècnica de Catalunya in Barcelona, Spain. His main research interests were Discrete, Computational and Combinatorial Geometry, and he was one of the main introducers of these subjects in Spain. He published more than a hundred papers in referred journals and about two hundred in conference proceedings, and he supervised fifteen Ph.D. theses. He was the lead for many national as well as international research projects. He served on several editorial boards of journals.

Enrique Molina was born in Madrid in 1989. He graduated from the Universidad Autónoma de Madrid in 2011 with a degree in Mathematics. He completed a Master's degree in Advanced Mathematics and Mathematical Engineering at the Universitat Politècnica de Catalunya in 2012. In 2013, he obtained a teacher training qualification and simultaneously started a Ph.D. in number theory at the Universidad Autónoma de Madrid.

Suneeta Ramaswami received a Ph.D. in Computer and Information Science from the University of Pennsylvania. Since 1997, she has been with the Department of Computer Science at Rutgers University-Camden, first as an assistant professor (1997–2004) and then as an associate professor. She has been a visiting scholar in the Department of Radiology at the University of Pennsylvania and at the Department of Applied Math at the Polytechnic University of Catalunya (Barcelona, Spain). Her main research interest is computational geometry. Her current work is on geometric techniques for three-dimensional surface mesh generation, adaptive meshing techniques for applications in medical imaging, and reconfiguration algorithms for self-organizing modular robots.

Vera Sacristán has a degree in Mathematics from the University of Barcelona, and a Ph.D. in Mathematical Sciences from the Universitat Politècnica de Catalunya. She currently is an associate professor at the Universitat Politècnica de Catalunya (Barcelona, Spain). She has been visiting scholar at the Department of Computer Science at the State University of New York at Stony Brook, and she regularly collaborates with several research groups, among which it is worth mentioning those at the Institute for Software Technology at the Graz University of Technology (Austria), the Institute of Computer Science at the Free University of Berlin (Germany) and the Computer Science Department at Rutgers University—Camden (USA). Her main research interest is in computational geometry. Her work focuses on a series of application problems related to proximity and more generally geometric graphs, visibility, location, and reconfiguration strategies for self-organizing modular robots.