Self-Reconfigurable Modular Robots and their Symbolic Configuration Space

S. Baarir[1], L.M. Hillah[1], F. Kordon[2], and E. Renault[2]

[1] LIP6, CNRS UMR 7606 and Université Paris Ouest Nanterre La Défense
200, avenue de la République, F-92001 Nanterre CEDEX, FRANCE
Lom-Messan.Hillah@lip6.fr, Souheib.Baarir@lip6.fr
[2] LIP6, CNRS UMR 7606, Université P. & M. Curie - Paris 6
4, place Jussieu, F-75252 Paris CEDEX 05, FRANCE
Fabrice.Kordon@lip6.fr, Etienne.Renault@gmail.com

Abstract. Modular and self-reconfigurable robots are a powerful way to design versatile systems that can adapt themselves to different physical environment conditions. Self-reconfiguration is not an easy task since there are numerous possibilities of module organization. Moreover, some module organizations are equivalent one to another.
In this paper, we apply *symbolic* representation techniques from model checking to provide an optimized representation of all configurations for a modular robot. The proposed approach captures symmetries of the system and avoids storing all the equivalences generated by permuting modules, for a given configuration. From this representation, we can generate a compact *symbolic configuration space* and use it to efficiently compute the moves required for self-reconfiguration (*i.e.* going from one configuration to another). A prototype implementation is used to provide some benchmarks showing promising results.

Keywords: Modular robotics, Self-reconfiguration, Symbolic configuration space, Symmetries, CKBot

1 Introduction

Context Modular robotics is an active research field where robots are assembled using numerous identical or different types of small modules. This is a powerful way to design versatile systems that can adapt themselves to different physical environment conditions or according to the purpose of their mission [9]. Moreover, self-reconfiguration allows to adjust the robot to a given task on the fly. Modular robots are thus perceived as a means to reach a balanced compromise between realization cost and multitasking capabilities. They are particularly well-suited to exploration (*e.g.* spatial) and search and rescue missions in hostile environments. Finally, they are robust and cheaper to produce.

Problem If auto-reconfiguration is a way to change the shape of a robot made of modules, it is also a way to create complex movements by means of successive configuration changes. The robot follows a path of configurations that allows it to move, to grab and drop objects, etc.

This is a key feature whose implementation faces numerous issues [12]:

– memory limitation, computation power and energy consumption,
– limited degrees of freedom, sometimes more constrained because some types of modules are more constrained than the others,
– coordinated communication between modules and inter-module communication schemes,
– structural symmetries in modules, generating equivalent modules configurations.

The last issue raises a problem for the computation of the *configuration space* for a robot composed of N modules. The size of the configuration space increases exponentially with N (where $N_i, i \in [1..T]$ when the robot is composed of T types of modules). Both the generation of the full configuration space and the identification of a given configuration in it are known problems [8].

Contribution The objective of this paper is to tackle the combinatorial explosion problem in the configuration space of modular robots. Our solution also helps to identify a given configuration among the ones that are equivalent. Self-reconfigurable modular robots are usually classified in two categories.

First, lattice-based self-reconfigurable robots can physically organize themselves in 2D or 3D grid structures. They are rigidly interconnected but are able to connect/disconnect and move relative to one another in a 2D or 3D space. In this kind of configuration, modules can only connect to their adjacent neighbors. Connection is assumed to be performed without alignment, because modules are assumed to be always aligned. This may not be true in practice for large configurations. The ATRON [3] is a typical example of a 3D lattice-based self-reconfigurable robot.

Second, chain-based robots can assemble in serial chains fashion (linear loops connections/disconnections) aligning themselves for connecting. They can form flexible configurations and are efficient for locomotion, since they can bend themselves in arbitrary angles to move. For instance, a snake-shaped robot can move like a snake because its modules bend in coordination to perform this kind of movement. The PolyBot [13] is a typical example of a chain-based robot.

This study focuses on CKBot [10]: a hybrid robot whose modules allow both chain and lattice reconfiguration capabilities [12].

The CKBot has two types of module: the UBar and the L7. Our study focuses on the UBar, whose picture you can find online at [10]. We only consider robots made of one type of module but this work can easily be extended to the case where several types of modules are involved.

Symmetries are a serious issue in the original explicit approaches to generate the CKBot configurations, when the number of modules grows. For instance, the methods presented in [8] take a lot of time for disambiguation because of the symmetries between numerous similar configurations. Our purpose is to propose a new approach for modeling the system, using symbolic representation techniques where symmetries are handled efficiently .

Our contribution lies in the following points. First, we propose an efficient symbolic representation of the modular robot configurations. It represents, by means of a single

matrix, both connections and orientations of modules. It is a significant improvement of the proposal made in [8] where two matrices are involved.

Then, we optimize the symbolic representation, where similar connection schemes of some inter-dependent connectors are only represented once in a symbolic way. From this symbolic representation, explicit configurations may be generated. Similarly to model checking, we produce the configuration space of the system as an oriented graph where nodes represent a set of equivalent configurations and arcs represent an action of any module in the system. This reduced graph replaces the traditional plain graph approach.

Therefore, we can exploit this configuration space to compute the moves that lead from a configuration c to any configuration $c' \in C$, the set of target equivalent configurations. This is a classical path search in an oriented graph.

Content Section 2 presents the CKBot UBar module and the two-matrix based representation technique of the robot configuration originally presented in [8]. We also describe the core principles of the symmetry techniques applied in this paper. Then, sections 3 proposes an alternative to the robot description of [8] and section 4 explains how we turn this new explicit representation into a symbolic one. Section 5 deals with the reconfiguration computation issue, where the transition system of the robot successive configurations is built and used to find paths between configurations. Finally, section 6 presents performance evaluation provided by a prototype implementation before a conclusion in section 7.

2 Problem Statement and Related Works

Controlling the configuration of modular and self-reconfigurable robots is computationally complex. This complexity depends on how the system is organized, both at the hardware and software levels. We consider the CKBot, where both a global bus and neighbor-to-neighbor communication schemes enable the system to determine its configuration.

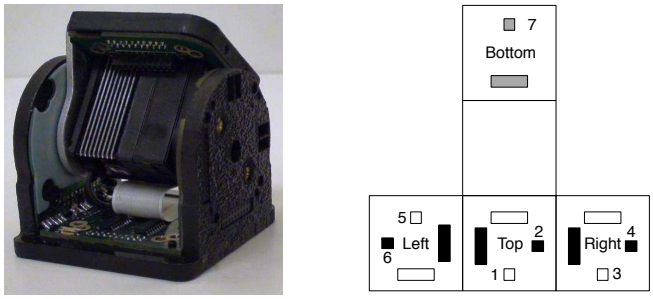
In the CKBot, connections between modules are represented by graphs, translated into adjacency matrices and ports adjacency matrices. Modules interconnect via connectors and are identified by values from 1 to N, where N is the number of modules[3]. Adjacency matrices are $N \times N$ matrices in which 1s denote interconnections between modules and 0s the absence of connection. In port adjacency matrices (which also are $N \times N$ matrices), non-zero numbers identify the IDs of the ports through which modules interconnect. Fig. 3 shows a CKBot assembly and its adjacency matrices.

We first present the CKbot module, then an explicit way to encode its configuration and the symmetry-based methods used in model checking to tackle combinatorial explosion. Section 3 applies this technique to efficiently encode the configuration space.

[3] Here, we only consider a system with one type of module. For a system with T types of module, we can consider the identities being $1,..,N_1,N_1+1,..,N_2,..,N_{T-1},..,N_T$ where N_i is associated to the i^{th} type of module, $i \in [1,T]$

2.1 Presentation of the CKBot

Fig. 1(a) presents a picture of a CKBot UBar module[4] and Fig. 1(b) shows its corresponding 2D schema representation. The CKBot UBar module has 7 ports (pair of infrared transmitters/receivers) and 20-pin headers on each of its 4 faces. Ports, represented by a square, are used for inter-module communication. 20-pin headers, represented by a rectangle, are used for electrical connection and communication on a CAN (Controller Area Network) bus. In the latter part of this paper, we may simply refer to couples ⟨port, 20-pin header⟩ as *connectors* (*e.g.* the Bottom face holds only one connector).



(a) A CKBot UBar module (b) Its corresponding 2D structure

Fig. 1. Overview of the CKbot

Top and Right faces share the same disposition for their connectors, whereas the one of Left is reversed (port 5). Thus, a module can attach to another in different ways. According to [12], each module can be *uniquely connected to another module in 10 ways (3 rotations for each of the 3 top faces and 1 orientation for the bottom)*. The only impossible connection is when two same faces are in front of one another in reverse positions. Fig. 2 presents an example of modular robot built from four CKBot modules[5].

[4] This picture is extracted from [8].
[5] This picture is extracted from [1].

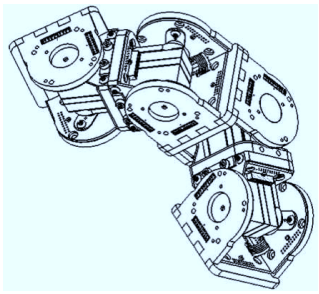


Fig. 2. A 5-module T shape robot and its corresponding adjacency matrices

2.2 Explicit Encoding of CKBot Configurations (from [8])

The technique used to describe the robot configuration is based on two $N \times N$ matrices, N being the number of modules[6]. The first matrix is a simple adjacency matrix, where 1-entries denote a connection between two modules and 0-entries no connection. The second matrix is a port adjacency matrix, where non-zero entries denote the type of connection from a module to another (referencing the port number). Fig. 3 shows a 5-module T shape and its corresponding adjacency matrices on the right.

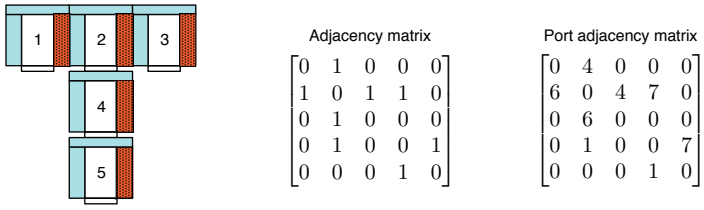


Fig. 3. A 5-module T shape robot and its corresponding adjacency matrices

This representation is suitable for configuration recognition on small configurations. However, it does not scale well. In particular, when reconfiguration is the problem under consideration, searching the configuration space for N modules which can move in parallel, quickly leads to combinatorial explosion of the configuration space for large values of N.

In [8], the author deals with automatic configuration recognition based on three principles:

- graph-based isomorphism identification: this method suffers from the exponential size of the automorphism group in the number of modules (worst case), as the size of the library of predefined configurations grows;
- port adjacency matrix spectral decomposition: it is very fast for small numbers of modules but suffers from numerical issues when numerous modules are involved. Explicit disambiguation due to symmetries in the configurations can be very long;
- heuristic-based linked list (called 3DLL) representation of the physical properties of configurations: it exploits the ports adjacency matrix of the modules. This method appears to be the most scalable, but suffers from the need to run exhaustively through every configuration in the library.

In order to tackle the combinatorial explosion in the configuration space, we propose to get inspiration from formal analysis methods [11] where this problem is common. Different and complementary techniques are used to reduce the size of the state space: decomposition, bounding, partial order, symmetry detection and the use of very efficient data structures (Decision Diagrams).

[6] In the remainder of this paper, N will always denote the number of modules that compose the complete robot.

It appears that the configurations of the CKBot can be organized into consistent sets of similar configurations where they only vary by module permutations. Hence, a more compact representation for the model can be designed to remove redundant and explicit information which can be inferred otherwise. In this setting, symmetry-based techniques are suitable to build their representation.

2.3 Compact Representation of Large State Spaces

This section presents through an example the principles of the symmetry-based techniques underlying compact representation of large state spaces in model checking. Formal definitions of the underlying theory can be found in [4].

Symmetry-based methods, exploit the presence of similarly behaving components to aggregate states (or, in our case, configurations) and state transitions (or, in our case, configurations changes) into equivalence classes. Hence, they generate a more abstract and compact state space: the *quotient graph*.

To present the quotient graph in the general framework, let us consider the classical example of a client/server system, with two identical clients C_1 and C_2 and a server S. Clients build a message $m \in \{m_1, m_2\}$, send it to S with their identity, and wait for an acknowledgment message. S processes incoming messages and then sends the acknowledgment to the client having issued the request.

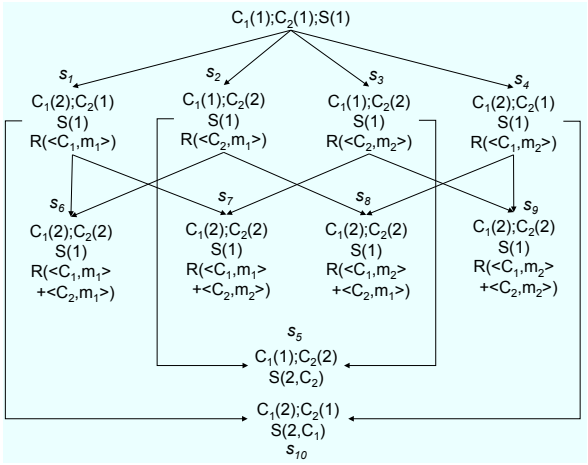


Fig. 4. First 11 states (among 24) of the reachability graph of the client/server example

We consider two local states for a client: (1) the message construction state and, (2) the receiving state. For the server, we consider: (1) the receiving state and (2) the sending of the acknowledgment. We also consider the network state R: $R(\langle C_1, m_1 \rangle)$ means that message m_1 of client C_1 is passing through the network R. Thus, the global state of our system will be the synthesis of all local states.

The behavior of the system can then be represented by *a reachability graph*, where nodes are global states, and arcs represent changes between states. Figure 4 represents

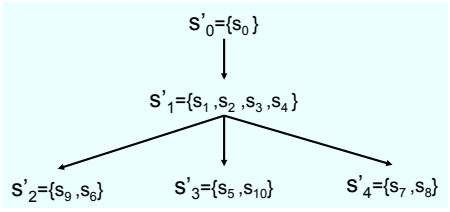


Fig. 5. First 5 states (among 10) of the quotient graph of the client/server example, w.r.t equivalence relation $\mathfrak{R}$

the beginning of the reachability graph of our toy example (the whole graph contains 24 states).

As shown in Fig. 4 the state space grows quickly with the number of clients and the type of messages. Exploitation of symmetries in the system helps to tackle this combinatorial explosion. We observe that C_1 and C_2 behave identically, hence they are symmetrical. Similarly, messages values are not distinguished by the server (*i.e.* they are processed identically), introducing another symmetry.

Let us formally identify these symmetries by an equivalence relation $\mathfrak{R} = \{C = \{C_1, C_2\}, M = \{m_1, m_2\}\}$, where C and M are equivalent classes. $\mathfrak{R}$ can then be used to build a quotient graph that preserves reachability and some temporal logic properties of the original reachability graph.

According to $\mathfrak{R}$, states of the reachability graph are partitioned in five equivalence classes: $s'_0 = \{s_0\}$, $s'_1 = \{s_1, s_2, s_3, s_4\}$, $s'_2 = \{s_6, s_7\}$, $s'_3 = \{s_5, s_{10}\}$ and $s'_4 = \{s_7, s_8\}$. The quotient graph corresponding to the 11 states of the reachability graph presented in Fig. 4 is represented in Fig. 5. Let us note that, in this case, its size neither depends on the number of clients nor the number of values for messages.

In the next section, we apply this technique to build all the configurations of a robot made with CKBot modules. To do so, there are three issues to deal with:

- identifying symmetries in the configurations in terms of permutations,
- elaborating an efficient symbolic representation for the equivalent classes generated by these permutations,
- building the symbolic transition relation.

3 Representing CKBot States

The first technique to fight against combinatorial explosion in the configuration space is to design a compact and efficient representation to model the robot. An important requirement for this compact representation is that it must preserve all the important information that cannot be computed from existing ones.

3.1 Matrix Representation

Our new model is also stored in a matrix, where 7 columns encode the ports connectivity of a module[7] and an 8^{th} column encodes its angle. Figure 6 shows the definition of this representation. The configuration of a module is thus encoded in a vector of size *Number of Ports + Degrees of Freedom*. The size of a configuration involving *N* modules is then *N*×*(Number of Ports + Degrees of Freedom)*. A CKBot module has 1 degree of freedom, hence a single column is sufficient to encode it.

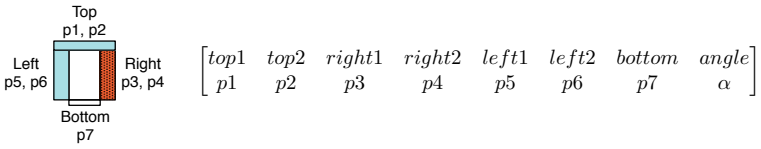


Fig. 6. Ports-connectivity matrix of an UBar module.

In the ports-connectivity matrix, non-zero entries denote a connection of the corresponding port (column) to a module whose identity is the value of the entry. As for the explicit representation (section 2.2), identities of modules range from 1 to *N*, so that no ambiguity is raised between an absence of connection and a module identity.

The angle scale ranges from 0 to 180° with a graduation in *D*° increments. To determine the physical position of a module from its representation as described in Fig. 6, angle is set to zero when the face *Bottom* is positioned such that ports 4, 6 and 7 are aligned. It is illustrated in Fig. 7, with *Bottom* oriented to the right. Angle 180 is determined when the module has made a rotation such that *Bottom* is oriented to the left.

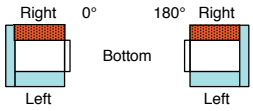
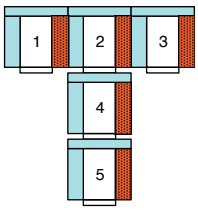


Fig. 7. Angles representation for a UBar module.

Example Let us determine the ports-connectivity matrix for the 5-module T shape example shown in Fig 3. The matrix encoding this configuration is shown in Fig. 8. It has the fixed 8 columns for the UBar and 5 lines for the number of modules. We can expect this type of representation to be much more compact than the adjacency matrices, where the number of modules will be in most cases greater than the sum of the numbers of ports and degrees of freedom.

[7] Numerotation of ports (*e.g.* $top1$, $top2$, etc.) refers to the 2D flat representation of a CKBot module as presented in Fig. 1(b).



$$
\begin{array}{l}
 \\
 \\
=> module1 \\
=> module2 \\
=> module3 \\
=> module4 \\
=> module5
\end{array}
\begin{array}{c}
\overset{top}{\longleftrightarrow}\ \overset{right}{\longleftrightarrow}\ \overset{left}{\longleftrightarrow}\ \overset{bottom}{\longleftrightarrow} \\
\begin{bmatrix}
p1 & p2 & p3 & p4 & p5 & p6 & p7 & \alpha \\
0 & 0 & 0 & 2 & 0 & 0 & 0 & 90 \\
0 & 0 & 0 & 3 & 0 & 1 & 4 & 90 \\
0 & 0 & 0 & 0 & 0 & 2 & 0 & 90 \\
2 & 0 & 0 & 0 & 0 & 0 & 5 & 90 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 90
\end{bmatrix}
\end{array}
$$

Fig. 8. Ports-connectivity matrix of the 5-module T shape of Fig. 3.

3.2 Matrix Encoding

An alphabet can be set up to encode all possible configurations of a module. Since a port can be connected or not, we need two values per port (for 7 ports). Thus, a 7-bit alphabet can encode this.

We set up an alphabet to encode all possibles configurations for a module. A port being either connected or not and since we have seven ports, we elaborate a compact alphabet to encode connectivity. Therefore, each letter in the alphabet is a concatenation of two parts:

- a first part contains 7 bits describing the connectivity of the described module, encoded as: $p1 \times 2^7 + p2 \times 2^6 + p3 \times 2^5 + p4 \times 2^4 + p3 \times 2^2 + p2 \times 2^1 + p1 \times 2^0$. Therefore, each configuration is represented in a unique way;
- a second part specifies the angle of the described module. There are x configurations per connectivity, where x is deduced from D ($x = 3$ when $D = 90°$, $x = 4$ when $D = 45°$, etc).

p1	**p2**	**p3**	**letter**
0	0	0	$X_{0,\alpha}$
0	0	1	$X_{1,\alpha}$
0	1	0	$X_{2,\alpha}$
0	1	1	$X_{3,\alpha}$

p1	**p2**	**p3**	**letter**
1	0	0	$X_{4,\alpha}$
1	0	1	$X_{5,\alpha}$
1	1	0	$X_{6,\alpha}$
1	1	1	$X_{7,\alpha}$

Fig. 9. Alphabet encoding the connectivity of a 3-port module. α represents the angle.

Example Let us consider a simple case with a module having only three ports. Figure 9 illustrates this encoding. There are $2^3 \times (\frac{180}{D} + 1)$ values in the alphabet. There is a total order since $\forall i \in [0, 2^3]$, $X_{i,\alpha} \leq X_{i,\beta}$ iff $\alpha \leq \beta$. For the CKBot, we thus use the following formula to compute the number of letters in the alphabet:

$$2^7 \times (\frac{180}{D} + 1) \tag{1}$$

4 Symbolic Representation of the CKBot Configuration Space

This section applies the symmetry technique described in section 2.3 to the representation presented in section 3. First, we identify equivalent configurations obtained from module rotation, then we discuss isomorphic configurations obtained by module permutation. We show how the two matrices can be combined and finally present a canonical way to express equivalence classes considering these two types of symmetries. This leads to the notion of *symbolic configuration space*.

4.1 Identification of Structural Symmetries

Module Rotation In the UBar CKBot module, there exists a symmetry between faces *Right* and *Left*. When this module is rotated with 180° in the orientation *Right-Left* (or in the reverse way), we obtain a mirror configuration. Ports 4 and 6 are symmetric, as well as ports 3 and 5. Therefore, a connection on p4 (or p3) with angle α is symmetric to a connection on p6 (or p5) with angle $(180 - \alpha) \mod 180$.

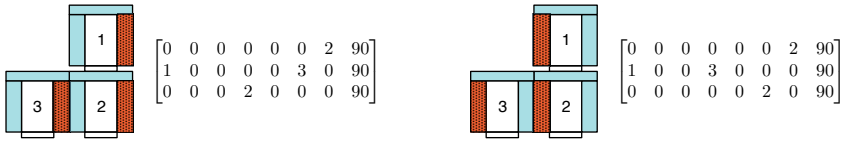


Fig. 10. Symmetry between two L-shape configurations

Figure 10 shows an example of symmetry in a 3-UBar CKBot configuration. On the left, the initial configuration. On the right, the symmetric one. We can observe that all modules have rotated. This corresponds to column permutation in the ports-connectivity matrices that are aside these configurations.

Module Permutation Two configurations are considered isomorphic when they form the same functional shape but where modules are permuted. Figure 11 shows an example of two T-shape isomorphic configurations with their ports-connectivity matrices. Modules have the same connectivity in the two configurations but modules 1, 2 and 3 are not in the same position. This corresponds to line permutation in the ports-connectivity matrix together with a value change to refer to the new modules id (the corresponding

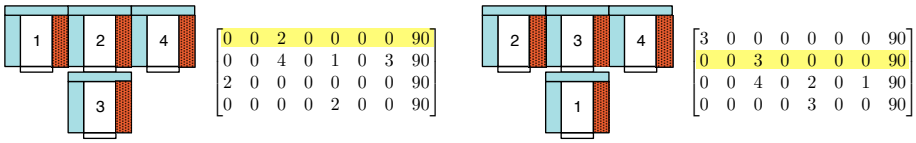


Fig. 11. Two T-shape isomorphic configurations.

lines are emphasized in the two matrices). On the first configuration, module 1 is on top left and is represented by the first line of the left matrix. On the second configuration, the top left module is 2 and is described by the second line in the right matrix. These two lines are identical in structure (third column is non-zero) but refer to different lines due to different neighbors.

Module Rotation and Permutation There is an issue to detect isomorphic configurations when they are also symmetric due to module rotation as shown in Fig. 12.

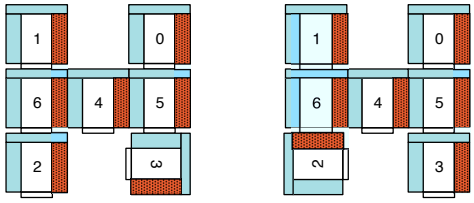


Fig. 12. Two H-shape isomorphic configurations.

Therefore, our symbolic encoding of our ports-connectivity matrix must integrate both types of equivalences together and distinguish all equivalence classes unambiguously.

4.2 Symbolic Encoding and Canonization

Encoding To encode the ports-connectivity matrix, we replace each line by its corresponding letter in the alphabet. Figure 13 shows, for module 1 to 5, (*i.e.* top to bottom) the encoding of a 5-UBar CKBot cross configuration (left) into an explicit ports-connectivity matrix (center) and then its corresponding symbolic representation (right). As mentioned in section 3.2, the number of letters in our alphabet is computed from formula (1).

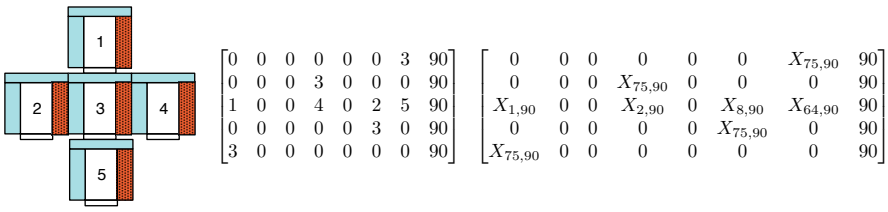


Fig. 13. Encoding of a 5-module cross shape configuration

In this figure, module 1 is connected to module 3 via port 7. So module 3 configuration is the value of the entry at line 1, column 7. Module 3 is connected to module 1 via port 1, so module 1 configuration is the value of the entry at line 3, column 1.

Canonization To compute the configuration space as a fixed point, we must compare symbolic states to detect if a new state has been already computed or not. However, the symbolic representation is not unique since it depends on the module order. We therefore need to canonize this symbolic representation for comparison purposes. Moreover, all representations of a given class of equivalent configurations must be computed from this canonical representation thanks to columns and lines permutations.

Since the alphabet we defined to encode the port-connectivity matrix is ordered, we can arrange the matrix by sorting lines as if there was a bit-encoding of integer values where 0 means 0 and non-zero values mean 1. We use the quick sort algorithm whose average complexity is in $n * Log(n)$.

Computed symbolic matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & X_{75,90} & 90 \\ 0 & 0 & 0 & X_{75,90} & 0 & 0 & 0 & 90 \\ X_{1,90} & 0 & 0 & X_{2,90} & 0 & X_{8,90} & X_{64,90} & 90 \\ 0 & 0 & 0 & 0 & 0 & X_{75,90} & 0 & 90 \\ X_{75,90} & 0 & 0 & 0 & 0 & 0 & 0 & 90 \end{bmatrix}$$

Canonized symbolic matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & X_{75,90} & 90 \\ 0 & 0 & 0 & 0 & 0 & X_{75,90} & 0 & 90 \\ 0 & 0 & 0 & X_{75,90} & 0 & 0 & 0 & 90 \\ X_{75,90} & 0 & 0 & 0 & 0 & 0 & 0 & 90 \\ X_{1,90} & 0 & 0 & X_{2,90} & 0 & X_{8,90} & X_{64,90} & 90 \end{bmatrix}$$

Fig. 14. Canonization of the ports-connectivity symbolic matrix shown in Fig. 13

Figure 14 shows the canonization of the symbolic matrix obtained in the example illustrated by Fig. 13. This operation only changes the order of lines. Then, any arrangement of modules can be considered by numbering lines with different module identities.

As an illustration, we can deduce from the canonical matrix, the cross configuration shown in Fig. 13 by labeling lines with module identities in the following order: 1, 4, 2, 5, 3. Similarly, all equivalent configurations due to module permutations can be reconstituted by setting new modules identities affected to lines (*e.g.* configuration where lines in the canonical matrix are 5, 4, 3, 2, 1 correspond to another configuration of the equivalence class).

Symbolic Configuration Space The symbolic encoding of the configuration space allows us to compute the *symbolic configuration space*. Each node of this reduced graph is a symbolic configuration. The symbolic configuration space is thus much smaller than the configuration space: the node ratio is exponential since each equivalence class grows with the number of modules (and only one symbolic configuration is required to store all the configurations that belong to this class).

In the next section, we focus on the way to compute the symbolic configuration space, as well as on the way to use it for defining the moves a robot made from CKBots must perform to change its configuration.

5 Reconfiguration

Reconfiguration of a modular robot is a key feature since it is used for both movement and adaptation. It faces both scalability and computation time issues, especially when

reconfiguration is to be performed on the fly. It is thus necessary to define an efficient transition system and operations. To do so, we take advantage of the symbolic representation elaborated in section 4.

5.1 Transition Relation Between Symbolic Configurations

A transition occurs when the robot changes from one symbolic configuration to another. More specifically, a transition occurs when a module changes its configuration in terms of connectivity and/or rotation. Several modules can perform a transition during a reconfiguration of the robot. We distinguish two types of transitions:

- **Functional Transitions:** they lead to rotations on the different modules degrees of freedom. They enable motion and do not alter the modules connectivity.
- **Structural Transitions:** they involve changes in modules connectivity and consist of a connection/disconnection of ports.

Functional Reconfiguration Functional reconfiguration does not alter the connectivity. It enables motion and involves the rotation of modules. In this setting, the functional successors of a module configuration are such that only the angle varies.

If D is the increment used on the angle scale such that $0 \leq D \leq 180$, let a module i which has a configuration $X_{i,\alpha}$. The possible successors of the current state for i are configurations $X_{i,\gamma}$ where $\gamma = ((\alpha + n \times D) \mod 180)$, $n \in \mathbb{N}$ (n being the number of D steps of the angle change).

All rotations cannot be performed in a functional reconfiguration. In particular, when all modules are interconnected as in Fig. 15, a rotation from 90 to 180 degrees cannot take place. Since all modules are connected, module 4 must first disconnect from module 1 before performing a rotation. Therefore, a set of structural transitions may be necessary before a functional reconfiguration can actually happen.

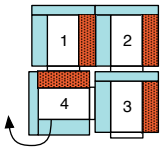


Fig. 15. Impossible rotation from 90 to 180 degrees for the bottom-left module

Structural Reconfiguration We consider in this paper that a functional reconfiguration only involves functional transitions, while a structural reconfiguration involves both kinds of transitions.

Since our study focuses on the CKBot, which is a hybrid robot (lattice or chain configurations), three assumptions must be made on the considered transitions:

Assumption 1: the successor of a symbolic configuration is reached through at most an atomic action for each module: connection/disconnection[8] or rotation.

Assumption 2: when a module is connected through one face only, it cannot disconnect, to avoid breaking the lattice or chain shape of the robot. This restriction is sometimes considered in similar work like [3].

Assumption 3: We consider that only one action is performed at a time for the N modules involved in the symbolic configuration[9].

Building Successors of a Symbolic Configuration To illustrate structural reconfiguration, let us consider again the simplified 3-port modules whose alphabet is presented in Fig. 9. Configuration $[010]$ (letter $X_{2,\alpha}$) has the following set of successors: $\{[010\ \gamma]$ (letter $X_{2,\gamma}$), $[011\ \alpha]$ (letter $X_{3,\alpha}$), $[110\ \alpha]$ (letter $X_{6,\alpha}$)$\}$.

5.2 Generating and Exploiting the Symbolic Configuration Space

Computation of the symbolic configuration space is similar to the generation of the state space in model checking. It corresponds to a fixed point on the exploration of all possibles moves. As for tools like greatSPN [7] that manage symmetries, each new symbolic state must be discovered by performing a symbolic evolution (*e.g.*, symbolic firing in model checking) of the system as described in [2].

Input: $Initial$, the initial configuration of the robot (encoded in a symbolic way)
Output: returns a symbolic configuration Space
$SymbConfSpace_1 = \emptyset$;
$SymbConfSpace_2 = Initial$;
while $SymbConfSpace_1 \neq SymbConfSpace_2$ **do**
  $SymbConfSpace_1 = SymbConfSpace_2$;
  foreach *configuration* $c \in SymbConfSpace_2$ **do**
    foreach *configuration* $s = Successor(c)$ **do**
      $s' = Canonize(s)$;
      if $s' \notin SymbConfSpace_2$ **then**
        $SymbConfSpace_2 \leftarrow SymbConfSpace_2 \cup s'$;
      end
      Add Link between c and s';
    end
  end
end
return $ConfSpace_1$;

Algorithm 1: Generation of the symbolic configuration space

[8] A module may connect or disconnect at most one face per reconfiguration.

[9] Similarly to model checking, we set an execution semantics at a low granularity: a single operation in the whole system. Interleaving between these single actions in the system ensures that we are compatible with a semantics where there are several parallel moves as for Petri Nets [6].

Algorithm 1 describes the computation of the symbolic configuration space. Successors of c are computed by applying possible connections/disconnections and angle rotation of each module of c.

Since the symbolic configuration space is an oriented graph, searching a move that leads from a concrete configuration c to any (the closest) concrete configuration $c' \in C$, the class of the target form the robot must reach is very easy. It corresponds to a shortest path search in an oriented graph like the Dijkstra algorithm [5] (in this case, all arcs in the symbolic configuration space are valued by 1).

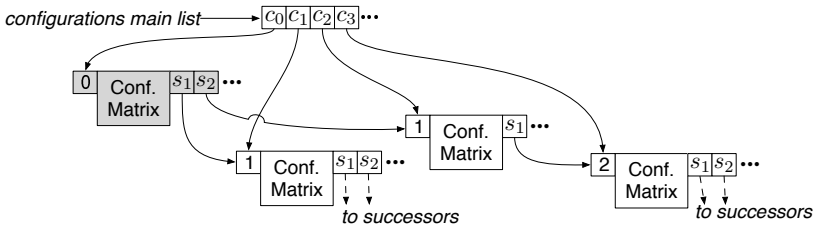


Fig. 16. Data structure to store the symbolic configuration space

So far, the data structure elaborated in the prototype is described in Fig. 16, which shows how 4 configurations $c_0, c_1, c_2, c_3 \ldots$ (out of more) are represented. *Configurations main list* represent the head pointer to this list of configurations. In this example, c_0 points to the initial configuration, which is shown in grey. Its distance to the initial configuration is 0 and it has two successors c_1 and c_2, which S_1 and S_2 of c_0 point to. The distance of c_1 and c_2 to the initial configuration is 1. We also show one successor of c_2 (c_3) whose distance to the initial configuration is 2.

Such data structure is suitable to search paths from one configuration to another one in the symbolic configuration space. Memory required to store the symbolic configuration space can be computed with formula (2) for the CKBot (7 ports and one degree of freedom):

$$Memory(bytes) = \overbrace{\underbrace{[(S_{int} \times N \times 8) + 1]}_{\text{one symbolic configuration}} \times NB_{sconf}}^{\text{all configurations}} + \overbrace{S_{pt} \times NB_{arcs}}^{\text{all successors}} + \overbrace{S_{pt} \times NB_{sconf} + 1}^{\text{main list}} \quad (2)$$

where N is the number of modules in the configuration, S_{int} is the number of bytes to store an integer, S_{pt} the number of bytes to store a pointer, NB_{sconf} the number of symbolic configurations in the state space and NB_{arcs} the number of arcs.

6 Performance Evaluation

To assess our modeling approach, we implemented a prototype to evaluate its benefits. The idea is to get an estimation of the gain provided by our symbolic representation. To do so, we consider two experiments:

- the construction of the symbolic configuration space to evaluate if it can be computed off-line and then embedded into a reasonable amount of memory,
- the on-the-fly computation of a path between a concrete current configuration of the system and the "closest"[10] concrete configuration in a class of configurations the system must reach.

For the experiments, we selected three types of initial configurations. First, the *line* configuration corresponds to a line of N modules. Second, the *square* configuration corresponds to a square of N modules. Finally, the *crawler* configuration is the repetition of a 4-CKBot pattern shown on the left side of Fig. 17.

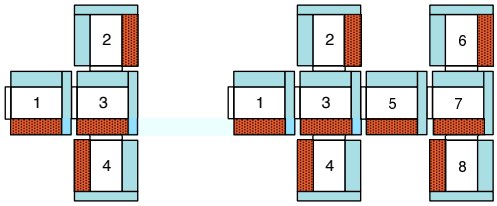


Fig. 17. Crawler configuration for 4 and 8 modules

All experiments to compute the symbolic configuration space were run on a 2.80GHz Intel Hyperthreaded Xeon computer with 14Gbytes of memory. As we show later, this does not mean that such a configuration is required to exploit the produced symbolic configuration space.

6.1 Experiment 1: Generating the Symbolic Configuration Space

For the first experiment, we computed the full symbolic configuration space from the three selected initial configurations with several values of N. For rotation, we consider D =90° (thus, three positions for the angle and an alphabet with $2^7 \times (\frac{180}{90} + 1) = 384$ letters according to formula (1)).

Table 1 summarizes the data collected in the first experiment. Columns, from left to right show: the value of N, the number of concrete configurations, the size of the symbolic configuration space (nodes/arcs), the ratio between the number of symbolic configurations and the number of concrete ones, the memory required by the program to compute the symbolic configuration space, the time required to compute the symbolic configuration space, and the estimated memory required to store the computed configuration space (evaluated with formula (2)).

[10] The one that can be reached with the smallest number of transitions.

N	Number of Concrete Configurations	Symbolic Representation: Size of Symbolic Config. Space	Ratio	Memory for Computation (MB)	Time for Computation (s)	Memory to Store Symbolic Conf. Space (MB)
Line configuration						
4	3888	81/202	$2\times4!$	3.44	0.42	0.019
6	$\sim 1.049\times10^5$	729/1822	$2\times6!$	12.48	9.39	0.159
8	$\sim 5.297\times10^7$	6561/16442	$2\times8!$	116.88	155	1.408
10	$\sim 4.285\times10^{11}$	59049/147622	$2\times10!$	1272	2496	12.619
Square configuration						
4^*	4032	84/925	$2\times4!$	4.45	0.56	0.064
8^*	$\sim 2.91\times10^8$	36093/388209	$2\times8!$	789	736.69	25.901
Crawler configuration						
4	3888	81/202	$2\times4!$	3.55	0.46	0.019
8^*	$\sim 1.01\times10^{10}$	124652/311360	$2\times8!$	2306	2286	26.616

Table 1. Evaluating performances of computation and storage of the configuration space

As expected, the symbolic configuration space offers great gains compared to the concrete one. The largest symbolic configuration space for 8 modules can be stored in a few mega bytes that is now easy to embed in small devices. This could not be the case with a concrete representation of the configuration space that quickly contains billions of elements. Moreover, symbolic ports-connectivity matrices are not stored as sparse ones and we use 64-bits integer to encode the alphabet and 64 bits to encode a pointer ($S_{int} = S_{pt} = 8$ bytes in formula (2)). Consequently, less memory could easily be consumed by using sparse matrices and/or less bits to encode a letter in the alphabet.

For *square* and *crawler* configurations, our fixed-point algorithm also computes configurations that must be discarded because they lead to deadlocks or absurd situations (*e.g.* modules that are located in the same tridimensional position). When such states are detected, we note the corresponding line with a * in Table 1. Such cases are usually rare compared to the size of the symbolic state space: for example, 360 configurations are discarded for the *square* with 4 modules, 88560 for the *square* with 8 modules and only 1 for the *Crawler* with 8 modules. This only affects the computation time and not our most important evaluation criterion in this experiment: the amount of memory required to store the symbolic configuration space.

We also note that we could not compute the configuration space for more than 8 modules in most cases. This is an implementation problem that should be considered in further work (intensive recursions and allocations lead to intensive memory consumption). Execution time is also quite long when N grows but, since the configuration space is computed off-line, there is no time constraint on this step of the process. This should be corrected later with a more refined version of our first prototype.

6.2 Experiment 2: Computing Paths in the Symbolic Configuration Space

For the second experiment, we computed the full configuration space from the three selected initial configurations with several values of N. Then, we performed several searches between a randomly selected departure concrete space and a randomly selected target symbolic state. We also considered $D = 90°$ for rotations.

Table 2 summarizes the data collected in this second experiment. Columns, from left to right show: the value of N, the size of the symbolic configuration state, the minimum,

average and maximum time (in ms) required to compute a path, the minimum, average and maximum length of computed paths.

N	Size of Symbolic Config. Space	Search Time (ms) Min	Avg	Max	Path length Min	Avg	Max
Line configuration							
4	81	0.018	0.0915	0.199	1	11.75	31
8	6 561	1.195	1.515	1.717	375	1 649	2 925
Square configuration							
4	84	0.021	1.107	1.697	0	17	88
8	36 093	0.506	1.212	404.87	19	781	2 604
Crawler configuration							
4	81	0.039	0.118	0.187	3	12	20
8	124 652	0.037	1.810	3.561	2	517	1 456

Table 2. Performances of configuration search

Once again, time performances are quite good and show that computation could be performed on the fly by a software that drives the robot since it never takes more than half a second (averages remains around a few milliseconds). The average size of computed paths remain reasonable, compared to the size of the configuration space (*e.g.* 781 transitions for a 10^{11} states configuration space in the case of the *Square* with 8 modules).

These data are extracted from a prototype that was not optimized and performances can clearly be enhanced.

7 Conclusion

Configuration recognition and dynamic auto-reconfiguration of modular and self-reconfigurable robots face numerous challenges, ranging from hardware to software and control issues. The one we focused on in this paper is related to the configuration space combinatorial explosion when the number of modules grows.

We present in this paper a symbolic encoding technique, inspired from the ones developed for model checking that also suffers from combinatorial explosion. We use this technique to represent configurations of the CKBot, an hybrid modular and self-reconfigurable robot. The symbolic representation of configurations exploits structural symmetries in the modules that allow to gather in one class several equivalent configurations.

These new techniques are far more efficient compact representation of the robot configuration space than the initial encoding presented in [8]. Modules rotation, permutation or a combination of both can be quickly recognized and disambiguated. Our technique can be easily adapted to robots presenting a hardware topology similar to the CKBot one. We call this new representation the *symbolic configuration space*.

A second contribution is the computation of reconfiguration. Since the symbolic configuration space can be stored in a small amount of memory, reconfiguration corresponds to the computation of a path between two nodes in the configuration space. This can be achieved by classical graph-based algorithms such as the one of Dijkstra.

Experiments made on a first prototype show promising results. The expected exponential gain between the symbolic configuration space of a system and its related configuration space is observed for several types of initial configurations. Thus, all configurations can be stored in a few MegaBytes that could not be the case otherwise (more memory can technically be embedded in small devices like the robot modules we consider). Computation of the symbolic state space can be long but this operation is typically performed off-line. Thus, no performances are really needed.

Experiment also showed that (still with our first prototype) a path between two configurations (*e.g.* reconfiguration of the robot) could be computed in a average time of a fewmilliseconds. This enables the use of our technique on the fly during the robot mission. When the robot is performing a move, the next one can be computed.

Since our experiment was done in a quickly implemented prototype, numerous optimizations can be applied that should increase performances.

Future work will consider some application to a real mission with a CKBot-based robots (possibly with several types of modules) to assess usability with several configurations when the robot is driven by an external computer or with embedded code.

References

1. D. Arney, S. Fischmeister, I. Lee, Y. Takashima, and M. Yim. Model-based Programming of Modular Robots. In *13th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'10)*, pages 87–91, Carmona, Spain, May 2010. IEEE Computer Society.
2. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic Well-Formed Coloured Nets for Symmetric Modelling Applications. *IEEE Transactions on Computers*, 42(11):1343–1360, November 1993.
3. D. J. Christensen. Evolution of shape-changing and self-repairing control for the ATRON self-reconfigurable robot. In *IEEE International Conference on Robotics and Automation, ICRA 2006.*, pages 2539–2545, 2006.
4. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry Reductions in Model Checking. In A. J. Hu and M. Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1998.
5. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, Cambridge, MA, second edition, 2003.
6. C. Girault and R. Valk, editors. *Petri Nets and System Engineering*, chapter 2, pages 9–23. Springer Verlag, 2003.
7. GreatSPN. Petri nets suite: `http://www.di.unito.it/~greatspn`.
8. M. G. Park. *Configuration recognition, Communication Fault Tolerance and Self-reassembly for the CKBot*. PhD thesis, University of Pennsylvania, 2009.
9. W.-M. Shen. Self-reconfigurable robots for adaptive and multifunctional tasks. Technical report, University of Florida, Florida, USA, Dec. 2008.
10. The CKBot home page. http://modlabupenn.org/ckbot/.
11. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, number 1491 in Lecture Notes in Computer Science, pages 429–528. Springer-Verlag, 1998.
12. M. Yim, P. White, M. Park, and J. Sastra. Modular Self-Reconfigurable Robots. In *Encyclopedia of Complexity and System Science*. Springer, 2009.
13. M. Yim, Y. Zhang, K. Roufas, D. Duff, and C. Eldershaw. Connecting and disconnecting for chain self-reconfiguration with PolyBot. *IEEE/ASME Transactions on mechatronics, special issue on Information Technology in Mechatronics*, 2003.