Modeling Robot Path Planning with CD++

Gabriel Wainer

Department of Systems and Computer Engineering. Carleton University.
1125 Colonel By Dr. Ottawa, Ontario, Canada.
gwainer@sce.carleton.ca

Abstract. Robotic systems are usually built as independent agents that collaborate to accomplish a specific task. Analysis of robot path planning consists of route planning and path generation. We will show how to apply the Cell-DEVS formalism and the CD++ toolkit for these tasks. We present a Cell-DEVS model for route planning, which, based on the obstacles, finds different paths available and creates a Voronoi diagram. Then, we show route planning using the Voronoi diagram to determines an optimal path free of collision. Finally, we introduce a Cell-DEVS model that can be applied to the routing of self-reconfigurable robots.

1. Introduction

The analysis of robot path planning in general include a multirobot system in cooperative environments (all mobile agents interact, trying to achieve a common goal). In most cases, the environment under study consists of a physical environment, a number of robots, objects in the environment, a set of predefined tasks, a task distribution scheme (specifying what to do at every moment), and intercommunication mechanisms. Path planning typically refers to the design of specifications of the positions and orientations of robots in the presence of obstacles. Path planning can be static or dynamic, depending on the mode in which the obstacle information is available. In order to follow the movement of robots in the work area, we need a spatial planner which must find a path free of obstacles to follow a predefined trajectory. In general, this consists of two phases:

- Route planning: a route is defined as a sequence of sub-goals that must be reached by the robots before reaching the final goal.
- Path generation: once the plan has been created, different heuristics (for instance, the shortest path) could be used to reach the predefined goal.

Cellular models provide an advantage to carry out these tasks. Route planning using Voronoi diagrams can be easily constructed using simple 2D cellular models (without needing to compute distance or intersections, sorting distances, and or explicit modeling of objects). Since cellular models only use local rules, any proposed algorithm can be applied to objects of arbitrary size/shape. Cell-DEVS [1] allows defining cell spaces using the DEVS (Discrete Events systems Specification) formalism [2] to define a cell space.

We present a Cell-DEVS model for route planning, which, based on the obstacles, finds different paths available and creates a Voronoi diagram. Then, we provide an algorithm for route planning, and we present an algorithm that takes the Voronoi diagram and determines an optimal path free of collision (considering the size of the robot). We apply this heuristics to create a Cell-DEVS model able to solve the route planning phase. Finally, we introduce an advanced Cell-DEVS model that can be applied to the routing of self-reconfigurable robots.

2. Background

Cell-DEVS improves execution performance of cellular models by using a discrete-event approach. It also enhances the cell's timing definition by making it more expressive. Each cell, defined as $TDC=< X, Y, S, N, delay, d, \delta_{INT}, \delta_{EXT}, \tau, \lambda, D >$, uses N inputs to compute its next state. These inputs, which are received through the model's interface (***X***, ***Y***), activate the local computing function (**τ**). State (**s**) changes can be transmitted to other models, but only after the consumption of a delay (**d**). Once the cell behavior is defined, a coupled Cell-DEVS is created by putting together a number of cells interconnected by a neighborhood relationship. A coupled Cell-DEVS is composed of an array of $t_1x\ldots xt_n$ atomic cells, defined as $GCC=< Xlist, Ylist, X, Y, n, \{t_1,...,t_n\}, N, C, B, Z >$. Each cell is connected to its neighborhood (N) through DEVS ports. Border cells (B) can have a different behavior or be "wrapped". Finally, the model's external couplings can be defined in the Xlist and Ylist. Each cell in a Cell-DEVS is a DEVS atomic model, and the cell space is a DEVS coupled model. DEVS is a formalism based on generic dynamic systems, including well defined coupling of components and hierarchical modular construction. A DEVS model is described as a composite of submodels, each of them being behavioral (atomic) or structural (coupled). Each atomic model, defined by $AM=< X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta>$, has an interface (X, Y) to communicate with other models. Every state (S) is associated to a time advance (ta) function, which determines its duration. Once this time is consumed, the model generates results by activating an output function (λ), and the internal transition function (δ_{int}) is fired. Input external events activate the external transition function (δ_{ext}). Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model's interfaces.

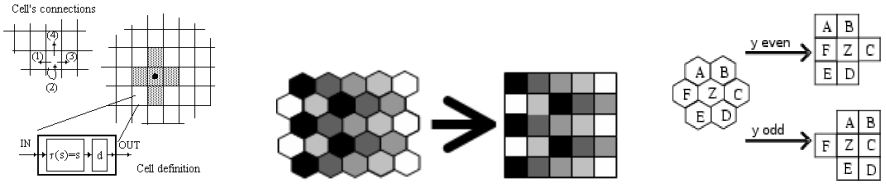

Fig. 1. Informal definition of Cell-DEVS, and shift mapping to the square lattice.

CD++ [3, 4] was developed following the definitions of the Cell-DEVS formalism. Cell-DEVS are described using a built-in specification language, which provides a set of primitives to define the diferent parameters of the model. The behavior of a cell (τ function) is defined using a set of rules of the form: *RESULT DELAY CONDITION*. When an external event is received, the rule evaluation process is triggered to calculate the new cell value. The CONDITION is evaluated; if satisfied, the new cell state is obtained by evaluating the RESULT expression. The cell will transmit these changes after a DELAY. A Lattice Translator allows using different topologies, which are translated into square CD++ rules, using the mechanism depicted in Fig. 1.

The algorithms here presented are based on Voronoi diagrams, which use the idea of proximity to a finite set of points in the plane P={*p1…pn*} (n ≥2). The diagram associates every point *pj* to their closest points *pi* (i ≠j), conforming covering sets [5, 6]. Points equidistant to two elements in P define the *border* of a region. The resulting sets define a tessellation of the plane (exhaustive, as every point belongs to a set, and they are mutually exclusive). Voronoi diagrams can be used to study the movement of a robot of a given size, describing paths surrounding the obstacles (and indicating the distance to them). These indicators allow a robot to determine if the path is feasible to pass through the path.

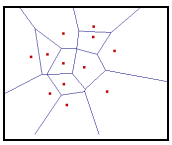

Fig. 2. Voronoi Diagram

We are also interested in models of self-reconfiguring robots. These systems are versatile in both their structure and the tasks they perform [7]. These robots are composed of a number of modules that can reshape according to the task to be carried out. Each robot is independent of the rest, and they act as parallel entities. The ability of reconfiguration leads to flow-based locomotion algorithms (allowing the robots to conform to the terrain on which they have to travel), which can be nicely modeled as cellular models.

3. Route Planning Models

Our path-planning model is based on [5] where CA are used to process a “top down” bitmap of a diamond-shaped area including a robot of arbitrary shape. The algorithm produces a Voronoi diagram that can be used to determine a path equidistant from obstacles in the space. Paths are calculated by marking the intersections of expanding “wavefronts” propagated by cellular expansion from given starting points. The input is an array of cells with values 1 (obstacle) or 0. The model executes in two stages:

1. Object boundary detection: cells and their neighborhoods are examined and compared to a set of 12 “edge code” templates. Each cell matching a configuration in the template uses the corresponding code (1-12) for the second stage.
2. Cells with edge codes are expanded in free space. Where expansions intersect, the cell of the intersection is given a timestamp and considered part of the final Voronoi diagram. The final state contains the Voronoi diagram.

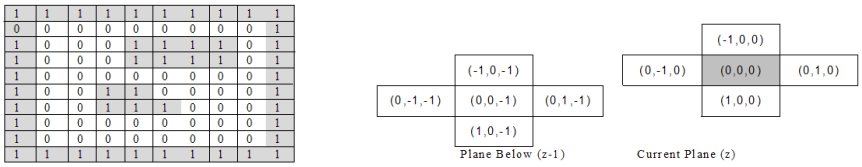

Fig. 3. (a) Input bitmap; (b) 3D neighborhood.

The following state variables are required for every cell: the *original encoding* of detected obstacles (0 or 1); the *calculated edge* code for the cell (1-12); a *flag* value used during the “wavefront expansion”, and the point on the *Voronoi diagram* representing this cell’s position. We put each state variable on a separated plane in a 3D Cell-DEVS. Plane 0 (x, y, 0) contains the original bitmap representing the space, Plane 1 (x,y,1) contains the edge codes, Plane 2 (x,y,2) includes the propagation of edge codes over time, and Plane 3 (x,y,3) stores the final Voronoi diagram. The 3D neighborhood is shown in Figure 3.b).

Fig. 4 describes the model definition in CD++. The model specification defines a 10x10x4 Cell-DEVS (a surface grid of size 10x10 and the four data planes). Four sets of rules which are used on each plane. The 3D cell model is effectively divided into four 2D models by using separate zones consisting of plane regions. The rule sets are:

– **nothing-rule:** used by the original data plane to keep the values from being changed.
– **bound-rule:** coding of edge directions. Patterns of cell values in each cell and its neighborhood are classified as one of 12 edge codes. The rules in this section perform the classification if the cells in the data plane correspond to one of 12 templates.
– **plane2-rule:** Cells with edge codes from 1-4 must be discarded. Cells with edge codes 5-12 are copied into a new grid and given a flag value for propagation in the third stage. The rules in this section carry over the values from the second plane which satisfy the criteria (4 < edge_code < 13).
– **plane3-rule:** the Voronoi diagram. In the previous plane, cells receive data values from their immediate neighbors and propagate the data out from any given starting point (points where these data wavefronts collide are those farthest away and equidistant from the starting obstacles; these are the points of interest when plotting a path for a robot). This plane examines the values in the plane below. If more than has its flag is set (and they do not contain the same values), the cell belongs to the Voronoi diagram. The Voronoi diagram is given the iteration number at which the cell was added to the diagram.

```
[Path-Finding]
dim : (10, 10, 4)  delay : transport localtransition : nothing-rule
neighbors:(-1,0,0)(0,-1,0)(0,0,0)(0,1,0)(1,0,0)(0,-1,-1) ... (0,1,-1)
zones : bound-rule { (0,0,1)..(9,9,1) } plane2-rule { (0,0,2)..(9,9,2) }
                  plane3-rule { (0,0,3)..(9,9,3) }
[nothing-rule]
rule: { (0,0,0) } 10 { t }
[bound-rule]
rule: 1 10 { (0,0,-1)=1 and (0,-1,-1)=1 and (-1,0,-1)=1 and (0,1,-1)=1
and (1,0,-1)=1 }
...
rule: 12 10 { (0,0,-1)=1 and (0,-1,-1)=1 and (-1,0,-1)=0 and (0,1,-1)=0
and (1,0,-1)=1 }
[plane2-rule]
rule: {(0,0,-1)+0.1} 10 { (0,0,-1) >4 and (0,0,-1)<13 }
...
rule: {(0,1,0)} 10 { fr((0,1,0))=0.1 and isint((0,-1,0)) and isint((-
1,0,0)) and isint((1,0,0)) }
[plane3-rule]
rule: {(time)} 10 { (0,0,0)=0 and %check and (-1,0,-1)!=(0,1,-1) }
...
rule: {(time)} 10 { (0,0,0)=0 and %check and (0,-1,-1)!=(0,1,-1) }
```

Fig. 4. Cell-DEVS model definition in CD++

The first example here presented shows the execution of the model using a partial boundary and two obstacles.

```
     +----------+    +----------+    +----------+    +----------+
   0|1111111111|   0|          |    0|          |   0|          |
   1|111       |   1| 57       |    1|          |   1|          |
   2|          |   2|   222222 |    2|          |   2|          |
   3|          |   3| 22222222 |    3|          |   3|          |
   4|          |   4| 222   22 |    4|          |   4|          |
   5|    111   |   5| 22 192 2 |    5|          |   5|          |
   6|    111   |   6| 22 857 2 |    6|          |   6|          |
   7|          |   7| 222   22 |    7|          |   7|          |
   8|          |   8|          |    8|          |   8|          |
   9|1111111111|   9|          |    9|          |   9|          |
     +----------+    +----------+    +----------+    +----------+
...
     +----------+    +----------+    +----------+    +----------+
   0|1111111111|   0|          |    0|          |   0|          |
   1|111       |   1| 57       |    1| 57777    |   1| 33  55   |
   2|          |   2|   222222 |    2| 57  92   |   2| 33 455   |
   3|          |   3| 22222222 |    3| 57 1922  |   3| 444444   |
   4|          |   4| 222   22 |    4| 5  192 2 |   4| 54 333   |
   5|    111   |   5| 22 192 2 |    5| 11119222 |   5| 54322234 |
   6|    111   |   6| 22 857 2 |    6| 88885777 |   6| 54322234 |
   7|          |   7| 222   22 |    7|  8 857 7 |   7|    333   |
   8|          |   8|          |    8|   88577  |   8|    444   |
   9|1111111111|   9|          |    9|          |   9|          |
     +----------+    +----------+    +----------+    +----------+
```

Fig. 5. Partial boundary and two obstacles

The inputs describe a boundary on the upper and lower horizontal edges of the 10x10 space, as well as two small obstacles inside the space. The input values in the first plane remain unchanged, and the edge codes in the second plane are generated after one iteration. The third plane is initially populated with edge codes >4, and these values are successively propagated across their neighborhoods (note the are "holes" where cells were out of reach of their neighbors). Propagation stops when cells have no more non-flagged neighbors. The final plane is the Voronoi diagram, where "for a diamond shape of diagonal size d, the path planning process selects those Voronoi edges that consist of points with labels of value $\ell \geq d+½$". Since the first values on the diagram are 2's, one should add that offset to find the desired values. In this case, for a robot of diagonal size 2, the points on the graph of value 4 or 5 represent viable travel paths which can be used by a robot of diagonal size 2 to travel avoiding the two obstacles.

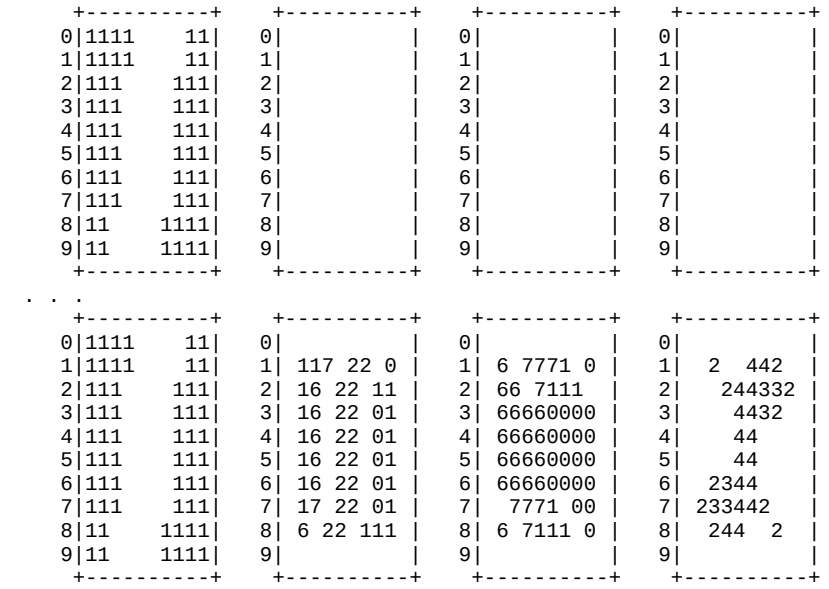

Fig. 6. Two large obstacles

Once we find the Voronoi diagram, we obtain a number of possible paths. We can find the shortest path based using a flooding technique like in [6]. We built a Cell-DEVS model to generate the shortest path: a cell is considered to be part of a valid path if its value is larger or equal to the robot size (*valid* cells). A cell with more than 2 valid neighbors is called a *node*. An *output node* is a cell where the robot is located before moving, and an *end node* is the destination. The shortest path to the end node is based on the Manhattan distance. The algorithm consists of two phases: flooding and selection. The *flooding* algo-

rithm explores all possible paths starting on the output node in parallel, choosing only valid cells. When a node is found, the path is divided in parallel. If during the exploration two paths are crossed, only the one with the best value continues. *Selection* starts when we get to the end node; we backtrack, looking for the minimum cost according to the chosen criteria. In this way, we can find a minimal path, as seen in the following figure.

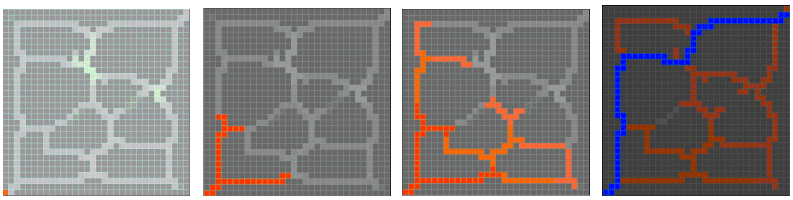

Fig. 7. (a) Initial Voronoi Diagram; (b-c) Flooding; (d) Selection

Our Cell-DEVS implementation encodes the distance to the objects at the beginning of the process (obtained by the Voronoi diagram selection presented in previous section). The following figure shows two examples of execution based on the original Voronoi diagrams. On Fig. 8.b), we see a modification, in which we have added an extra connection in the bottom-left part of the diagram (which affects the shortest path found).

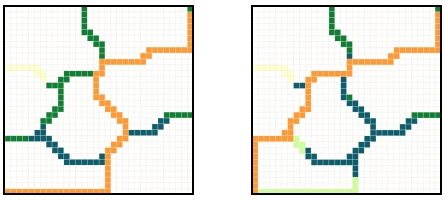

Fig. 8. (a) Shortest path (b) Shortest path with modified Voronoi diagram

4. Modeling self-reconfiguring robots

In this section we will show how to model self-reconfiguring robots, based on the work presented in [7]. Self-reconfiguring robots are composed identical modules that can autonomously reshape. The problem we will use as our case study is that of robotic locomotion in the two-dimensional plane, following a flow-like locomotion pattern. The model is capable of: (1) linear motion on plane of modules; (2) convex transitions into a different plane; and (3) concave transitions into a different plane [1]. The control algorithm uses local rules and it is constructed as a cellular model. We will show the behavior of a self-reconfiguring robot moving in a non-structured space, avoiding the obstacles presented.

Ten different states can be defined for each cell: empty (0), occupied by a non-moving module (1), occupied by an obstacle, or occupied by a robot moving in N/S/E/W direction (3-9). The model uses a modified Moore Neighborhood. The model consists of 27 rules controlling the full behavior of a cell. Each cell can be in a specific state, from a total of 10 states. The basic idea behind the model is that locomotion is produced from a two-phase mechanism, in which in the first phase each cell determines if it has to change its state, and the new state it will reach. On the second phase, depending on the state of each neighbor, a cell might decide to cancel its decision, or to go ahead as planned.

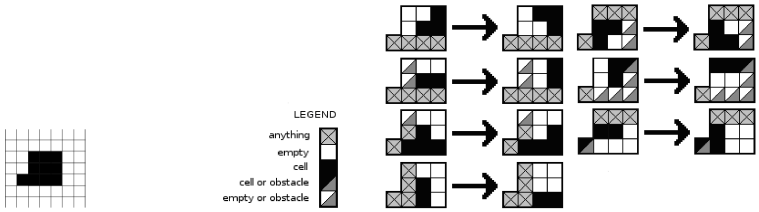

Fig. 9. (a) Neighborhood shape; (b) Model's rules.

The following figure shows some of the execution results obtained when using a square topology. Particularly noteworthy is the fact that the robot climb obstacles with a relative height of 3 units, and when it climbs down, it follows the shape of the terrain.

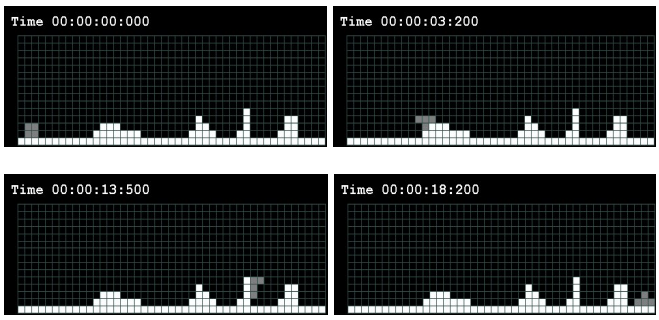

Fig. 10. Model execution.

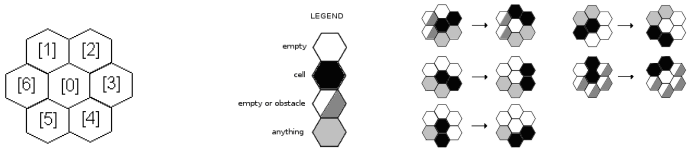

Fig. 11. (a) Hexagonal Neighborhood definition; (b) Model's rules.

The model was extended to a hexagonal topology, resulting is on the same two-phase mechanism, but fewer rules (21). The amount of possible states is also reduced (8). The following figure shows a graphical representation of the model, showing the local rules. The following figure shows the model representation using hexagonal Cell-DEVS in CD++ (notation in Fig. 7a)

```
[reconfig-robot-hexa]
dim : (15,45)              delay : transport   border : wrapped
neighbors : (-1,-1)(-1,0)(-1,1)(0,-1)(0,0)(0,1) (1,-2) (1,-1) (1,0) (1,1)

[reconfig]
rule: 1 100 {[0]=0 and [4]=1 and [5]=3 and ([6]=0 or [6]=2)}
rule: 1 100 {[0]=3}
rule: 4 0 {[0]=1 and [1]=0 and [2]=0 and [3]=0 and [4]=1}
rule: 0 100 {[0]=4 and [1]=0 and [2]=0 and [3]=0 and [4]=1}
rule: 1 100 {[0]=0 and [1]=0 and [5]=1 and [6]=4}
rule: 1 100 {[0]=4}
rule: 5 0 {[0]=1 and [1]=0 and [2]=0 and [3]=0 and [4]=0 and [5]=1}
rule: 0 100 {[0]=5 and [1]=0 and [2]=0 and [3]=0 and [4]=0 and [5]=1}
rule: 1 100 {[0]=0 and [1]=5 and [2]=0 and [6]=1}
rule: 1 100 {[0]=5}
...
```

Fig. 12. (a) Hexagonal Neighborhood definition; (b) Model's rules.

The following figure shows the model's execution. As we can see, the results obtained are similar to those presented in Figure 12, using the hexagonal topology. Nevertheless, using a square topology required 18.2 seconds to travel across all obstacles, while the second robot, modeled with a hexagonal topology required only 15.8 seconds.

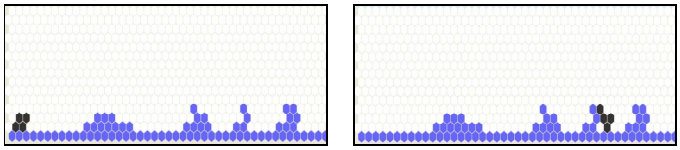

Fig. 13. Model execution.

5. Conclusion

We have introduced the use of CD++ for applications of path planning in robotic applications. We first presented a model that correctly simulates the behavior of path-finding algorithms, creating a Voronoi diagram as a result. The map describes paths surrounding the obstacles, and indicating the distances between them, allowing determining if a robot can pass through the path. After, we presented an algorithm that takes the Voronoi map

and determines a shortest path between the robot and the destination. The use of hexagonal topology, with fewer rules, resulted in faster movement. The cellular models presented show the feasibility of this approach in solving complex application using very simple rules, permitting observation of emerging behavior. In this way, one can develop algorithms that can execute parallel searches and improve the quality and speed in the determination of the paths.

The use of cellular models is very efficient, as it can operate extremely quickly (in just a few cycles of evolution) and every cell is being solved in parallel, in contrast to more traditional, mathematical approaches which require more complex calculations of distances and angles. The downside is that it does require a full knowledge of the obstacle. In addition, the model does not provide a complete solution in the case where there is not one distinct solution path.

Acknowledgments

This work was partially funded by Precarn and NSERC. Different students participated in developing the models here presented: Javier Ameghino, Alejandro Baranek, Ricardo Kirkner, Kevin Lam, Maximiliano Polimeni and Marcela Ricillo.

References

1. Wainer, G., Giambiasi, N.: N-dimensional Cell-DEVS. Discrete Events Systems: Theory and Applications, Kluwer, Vol.12. No.1 (2002) 135-157.
2. Zeigler, B., Kim, T., Praehofer, H.: Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press (2000).
3. López, A., Wainer, G. Improved Cell-DEVS model definition in CD++. In: P.M.A. Sloot, B. Chopard, and A.G. Hoekstra (Eds.): ACRI 2004, LNCS 3305. Springer-Verlag. 2004.
4. Wainer, G.: CD++: a toolkit to develop DEVS models. Software - Practice and Experience. vol. 32, pp. 1261-1306. (2002).
5. P. Tzionas, A. Thanailakis and P. Tsalides. Collision-Free Path Planning for a Diamond-Shaped Robot Using Two-Dimensional Cellular Automata", IEEE Trans. On Robotics and Automation. Vol. 13, No. 2. pp. 237-246 (1997).
6. Behring, C., Bracho, M. Castro, M., Moreno, J. A.: An Algorithm for Robot Path Planning with Cellular Automata. Proceedings of ACRI 2000. Karlsruhe, Germany. (2000).
7. Butler, Z., Kotay, K., Rus, D., Tomita, K.: Generic Decentralized Control for a Class of Self-Reconfigurable Robots. Proceedings of 2002 IEEE International Conference on Robotics and Automation, ICRA 2002. Washington, DC, USA. (2002).