INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 13(6), 1007-1031, December 2018.

Simulation of Rapidly-Exploring Random Trees in Membrane Computing with P-Lingua and Automatic Programming

I. Pérez-Hurtado, M.J. Pérez-Jiménez, G. Zhang, D. Orellana-Martín

Ignacio Pérez-Hurtado*, Mario J. Pérez-Jiménez, David Orellana-Martín
Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville, Seville, Spain
{perezh,marper,dorellana}@us.es
*Corresponding author: perezh@us.es

Gexiang Zhang
School of Electrical Engineering
Southwest Jiaotong University, Chengdu, Shichuan, China
zhgxdylan@126.com

Abstract: Methods based on Rapidly-exploring Random Trees (RRTs) have been widely used in robotics to solve motion planning problems. On the other hand, in the membrane computing framework, models based on Enzymatic Numerical P systems (ENPS) have been applied to robot controllers, but today there is a lack of planning algorithms based on membrane computing for robotics. With this motivation, we provide a variant of ENPS called Random Enzymatic Numerical P systems with Proteins and Shared Memory (RENPSM) addressed to implement RRT algorithms and we illustrate it by simulating the bidirectional RRT algorithm. This paper is an extension of [21][a]. The software presented in [21] was an ad-hoc simulator, i.e, a tool for simulating computations of one and only one model that has been hard-coded. The main contribution of this paper with respect to [21] is the introduction of a novel solution for membrane computing simulators based on automatic programming. First, we have extended the P-Lingua syntax –a language to define membrane computing models– to write RENPSM models. Second, we have implemented a new parser based on Flex and Bison to read RENPSM models and produce source code in C language for multicore processors with OpenMP. Finally, additional experiments are presented.
Keywords: Membrane Computing, RENPSM, robotics, RRT, P-Lingua.

[a]Reprinted (partial) and extended, with permission based on License Number 501431225 ©[2018] IEEE 7th International Conference on Computers Communications and Control (ICCCC)

1 Introduction

Robots are machines oriented to objectives equipped with actuators, sensors and computation units acting under physical constraints. Regardless of their morphology, they should accomplish tasks by acting in the real world. This is one of the main reasons by which robot motion planning is an eminent research area in robotics [8, 11, 21]. In general terms, the problem of motion planning can be defined in the configuration space of a robot as follows: *Given a start configuration state, a goal configuration state, a geometric description of the robot, and a geometric description of the environment, find a path that moves the robot gradually from start to goal.*

A configuration state is a specification of the positions of all robot points relative to a fixed coordinate system. This is usually expressed as a vector of positions and orientations, for example, a rigid-body robot in a 2D world can be expressed as a vector (x, y, θ) representing the

center (x, y) of the robot in a fixed coordinate system and its yaw angle θ, i.e, the heading angle of the robot. Since the shape of the robot is described, all of its points are then known.

Several constraints can be added to this problem, the most common is to reach the goal while never touching any obstacle in the environment. Others can also be added, for example, a social robot could restrict configuration states in order to guarantee the human comfort.

The configuration space of a robot can also be constrained by the type of movements the robot can perform. In this sense, nonholonomic robots are those that cannot instantly modify its direction without employing rotation in-place. On the other hand, holonomic robots can do it (assuming zero mass). For example, a holonomic robot in a 2D world can move along the x axis and the y axis, as well as modify its yaw angle if needed. But a nonholonomic robot can only move forward/backward and/or modify its yaw angle. This is the typical case of dual-wheeled mobile robots and cars.

Classical path planning algorithms have been widely adapted and applied to the problem of motion planning with constraints in robots, for example, in [26], an application of the Dijkstra algorithm for robot path-planning was presented. In such solutions, the general problem is usually divided into two smaller problems: the *global path planning* problem, as described above; and the *local path planning*, where the robot tries to connect two consecutive states in real-time considering features not included in the global plan as, for example, dynamic obstacles. The accumulated error during the local planning conducts to periodically recompute the global plan. For this reason, the computational complexity of global planners is a critical point regarding to real-time constraints. Many efforts have been made to provide good global planners. For example: in [25], a search algorithm, called D^*, was presented for path planning in real-time environments. In [15], a variant of the classical search algorithm A^* is applied to grids with blocked and unblocked cells. In [12], a tool for global path planning, called Rapidly-Exploring Random Trees (RRT), was presented.

The class of RRT algorithms for global path planning is based on the randomized exploration of the configuration space before moving the robot by building a tree in memory where nodes represent states that can be reached by the state of the corresponding parent in a fixed amount of time, furthermore each edge contains a velocity reference to reach the state in the child node from the state in the parent node. It is currently one popular method in robot motion planning due to its good properties. The computed RRT can be used together with search algorithms or, as presented in [13], the RRT generation algorithm can be used by itself as a path planning algorithm, where two RRTs are built simultaneously, one beginning from the initial configuration and another one beginning from the ending configuration (*bidirectional RRT*).

In order to follow the path in safe manner, a local planner module should be executed considering dynamic obstacles. Finally, each motor of the robot must be able to reach and maintain velocity references for fixed periods of time. This is the function of a type of software called *controller* on-board of the robot. Thus, *robot control* [1] is the branch of robotics dedicated to the study and practice of controlling robots.

Robot controllers are usually based on common silicon microprocessors, but in the recent years, some classes of *membrane systems* [16] have been in use for modelling them [18] [19] [20] [29]. Membrane systems are models of computation based on the structure and functions of the living cells. In a membrane system, there are objects being evolved inside compartments according to rules applied in a non-deterministic, maximally parallel way. They have been used as a new technique to attack the P versus NP problem [22], and several applications have been also studied: stochastic P systems for modelling biological phenomena [24], probabilistic P systems for modelling real ecosystems [2], spiking neural P systems incorporating fuzzy reasoning, for fault diagnosis and learning [27], and others.

With respect to robot control, numerical P systems (NPS) were used for modelling and sim-

ulating robot controllers [20], although the initial application of NPS was related to economical processes [17]. A variant called *enzymatic numerical P systems* (ENPS) [18] was introduced and applied to the distributed control of a swarm of mobile robots. Indeed, reactive and proportional-integral-derivative (PID) dual-wheeled robot controllers have been successfully designed and simulated by means of ENPS, as well as software simulation tools [29]. This variant has been also used [19] to address *robot localization* problem [8], where the robot must know its position in the environment by using sensors.

In [21], following [23], a new variant of ENPS called *random enzymatic numerical P systems with proteins and shared memory* (RENPSM, for short) was introduced. New syntactical ingredients were included to fit the requirements of the RRT algorithm:

- *Random numbers*: The algorithm uses a randomized method to explore the physical space, therefore random numbers must be generated.
- *Shared memory*: The algorithm is parallelized using processes sharing common variables, and a distinguished membrane, called *shared memory*, is included. At any instant, each membrane can read from or write to it.
- *Proteins*: In order to synchronize the sequential execution of the algorithm, proteins are used.

This paper is an extension of [21]. The software presented in [21] was an ad-hoc simulator, i.e, a tool to simulate computations of one and only one model that has been hard-coded. Ad-hoc simulators can be optimized for specific hardware architectures, but they are less debug-friendly than generic simulators, since changes in the model imply changes in the source code of the simulator. On the other hand, P-Lingua [3, 7, 33] is a language to define membrane computing models, there are several simulators based on P-Lingua, most of them are implemented in the pLinguaCore library [33] in Java language. In this paper, we provide a novel approximation taking the advantages of ad-hoc and generic simulators by using automatic programming. We have implemented a tool for parsing P-Lingua files defining RENPSM models and generating source code in C and OpenMP for ad-hoc simulators. Thus, we have a flexible way to debug since we are using a language to define the models instead of hard-coding them in the source code. Moreover, the generated source code is able to run on multicore processors by using OpenMP. Furthermore, additional virtual experiments are presented in this paper.

This paper is structured as follows. In the next section, some notions about robot path planning are introduced. In Section 3, the rapidly-exploring random trees (RRTs) are described with some details. Section 4 is devoted to introduce random enzymatic numerical P systems with proteins and shared memory. In Section 5, a RENPSM model for the bidirectional RRT algorithm is described. In Section 6, the original software presented in [21] based on C++ and ROS [30] is explained, including some experimental results. In Section 7, the new software presented in this paper is introduced, including an extension of the P-Lingua syntax for RENPSM, as well as additional experiments. Finally, conclusions and future work are drawn.

2 Robot path planning

In general terms, robot path planning can be solved by applying a solution based on three modules:

- *Global planner*: It receives the desired ending configuration of the robot, its safety radius and current localization, as well as the precomputed position of the static obstacles in

the environment and current information of sensors and odometry. The current dynamic obstacles detected by the sensors are added to the static ones in order to generate a more descriptive information of the environment. The odometry is used to obtain the current velocity of the robot when kinodynamic constraints are considered. Then, the global planner computes a plan from the starting configuration x_{init} to the desired final configuration x_{end} of the robot. The plan is represented as a sequence of local goals $\{g_i | 1 \leq i \leq n\}$, where $g_1 = x_{init}$ and $g_n = x_{end}$. Each goal can be reachable from the previous goal considering the constraints of the problem, i.e, avoiding static obstacles, nonholonomic and kinodynamic constraints, etc. RRT algorithms and other similar algorithms can be used for this task.

- *Local planner*: It receives the sequence of local goals generated by the global planner, as well as the current information of sensors, localization and odometry, then it sends velocity references to the controller in order to command the robot along the path. Several algorithms such as *the dynamic window approach* [5], *pure pursuit* [4], and *potential fields* [10] algorithms, among others and variants, can be used.

- *Controller*: It receives velocity references from the local planner and manages the power of the motors to fit each reference and maintain it constant or accelerate or reduce it until the next one.

In Figure 1, it is represented the general robot path planning cycle. First the robot computes a global plan from its current pose to the desired ending pose; if the plan can be generated, i.e, the robot could reach the destination considering all the constraints, then the local planner receives a sequence of intermediate goals and sends velocity references to the PID controller in order to follow the path in a safe manner until reaching the destination; if some error occurs, for example, a dynamic obstacle is too close to the next local goal or the robot is too far from the next local goal (considering a predefined threshold), then the global plan is recomputed from the current robot position. Notice that if there is a dynamic obstacle too close to the ending configuration, then the global plan cannot be found.

3 Rapidly-exploring Random Trees

An RRT [12] is a randomized tree structure for rapidly exploring in memory a *state space* X from an initial state x_{init}. It can be successfully used for nonholonomic and kinodynamic global path planning in robotics [13].

Nodes in an RRT represent possible reachable states, for mobile robots in a 2D world which is given by (x, y, θ) where (x, y) are the Cartesian coordinates of the robot position and θ is the heading angle. However, the heading angle can be omitted in order to reduce the size of the tree.

It is assumed that a fixed *obstacle region* $X_{obs} \subseteq X$ must be avoided, so the nodes of the RRT are states in X_{free}, the complement of X_{obs} in X.

Edges in an RRT represent transitions between reachable states, each of which is labelled with the velocity reference u that the robot should execute for a fixed period of time Δt in order to change the corresponding states. For a mobile robot in a 2D world, the velocity reference can be represented by the pair of linear and angular velocities (v, ω) to be sent to the controller. On the other hand, if θ has been omitted, the edges in the RRT are labelled with instant linear velocities. Thus, a holonomic robot can reach a state x_1 from another state x_0 connected by an edge labelled with u by applying $x_1 = x_0 + u \cdot \Delta t$. In the case of nonholonomic robots, the local planner should select the best sequence of motions in order to approximate x_1, for example, if

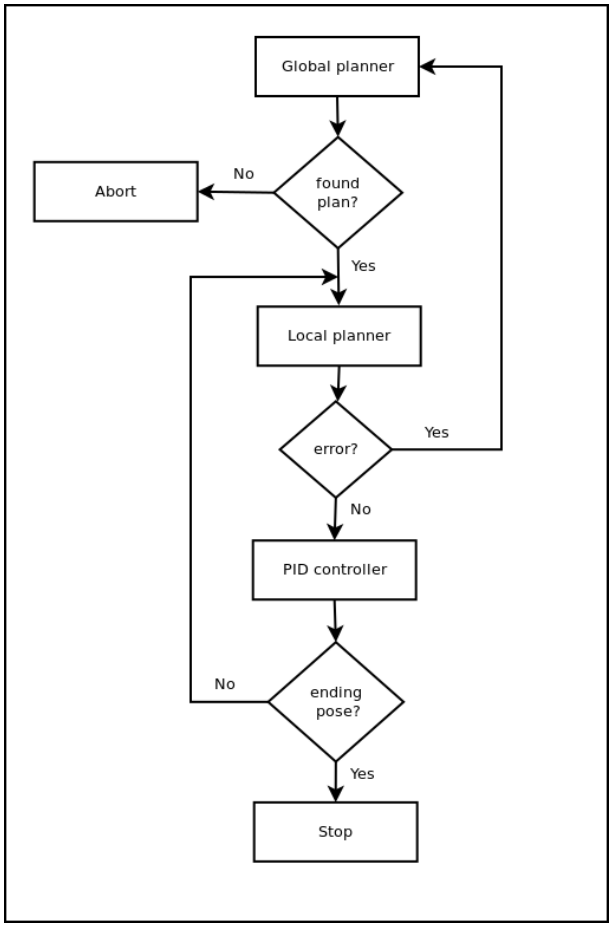


Figure 1: Robot path planning cycle

the robot can rotate in-place, a naive solution is to perform a rotation in-place before developing the motion in a straight line.

Algorithm 1 GENERATE_RRT

Require: $x_{init}, K, \rho, \Delta t, X, X_{obs}, d_{min}$
1: $V_\tau \leftarrow \{x_{init}\}$; $E_\tau \leftarrow \emptyset$;
2: **for** $k = 1$ to K **do**
3: $x_{rand} \leftarrow$ RANDOM_STATE(X);
4: EXTEND($\tau, x_{rand}, \rho, \Delta t, X_{obs}, d_{min}$);
5: **end for**
6: **return** $\tau = (V_\tau, E_\tau)$

Algorithm 1 is an iterative method to generate an RRT using the function EXTEND defined in Algorithm 2, where:

- x_{init} is the initial state.
- K is the number of iterations to build the RRT.
- ρ is a prefixed distance metric.
- Δt is a fixed amount of time for transitions.

Algorithm 2 EXTEND

Require: $\tau, x, \rho, \Delta t, X_{obs}, d_{min}$
1: $x_{near} \leftarrow$ NEAREST_NEIGHBOR(x, τ);
2: **if** DISTANCE$(x, x_{near}) \geq d_{min}$ **then**
3: $u \leftarrow$ SELECT_INPUT(x, x_{near});
4: **if** $\neg$ COLLISION$(x_{near}, u, \Delta t, X_{obs})$ **then**
5: $x_{new} \leftarrow$ NEW_STATE$(x_{near}, u, \Delta t)$;
6: $V_\tau \leftarrow V_\tau \cup \{x_{new}\}$
7: $E_\tau \leftarrow E_\tau \cup \{(x_{near}, x_{new})\}$
8: **if** DISTANCE$(x, x_{new}) < d_{min}$ **then**
9: **return** *Reached*;
10: **else**
11: **return** *Advanced*;
12: **end if**
13: **else**
14: **return** *Trapped*;
15: **end if**
16: **else**
17: **return** *Trapped*;
18: **end if**

- X is the state space.
- X_{obs} is the obstacle state space.
- d_{min} is the minimum distance threshold according to ρ in order to include a new node in the RRT.
- $\tau = (V_\tau, E_\tau)$ is the RRT generated.
- RANDOM_STATE(X) is a function to get a random state from X
- NEAREST_NEIGHBOR(x, τ) is a function to get the closest node to x in τ according to ρ.
- DISTANCE(x, x_{near}) is a function to get the distance of x to x_{near} according to ρ.
- SELECT_INPUT(x, x_{near}) is a function to get the velocity input that should be commanded to the robot in order to achieve state x from x_{near}.
- COLLISION$(x_{near}, u, \Delta t, X_{obs})$ is a function returning *true* if a collision could be produced moving the robot from state x_{near} by applying the input u for Δt time considering the obstacles in X_{obs}.
- NEW_STATE$(x_{near}, u, \Delta t)$ is a function to get a new state x_{new} by applying the input u to the robot for Δt time starting at state x_{near}.

The function EXTEND tries to add a new node to the RRT τ considering a reference x. If the function fails, then it returns *Trapped*; if the new node is closer than d_{min} to x, then it returns *Reached*; and if the new node is far from x considering d_{min}, then the function returns *Advanced*. Figure 2 describes a RRT generated after 5000 iterations by using Algorithm 1 with the Euclidean distance and omitting the heading angle.

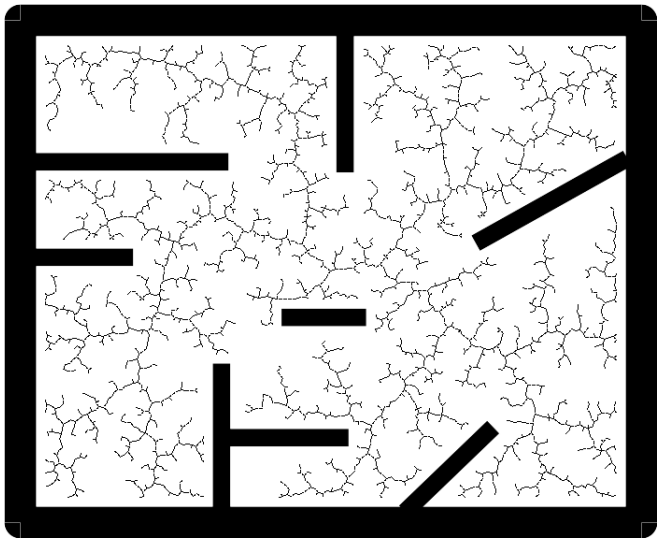


Figure 2: RRT generated after 5000 iterations

In [13], a bidirectional RRT algorithm was introduced for path planning. The main idea of this algorithm is to create two RRTs: τ_a starting at x_{init} and τ_b starting at x_{end}. If τ_a and τ_b are connected in a prefixed number K of iterations, then a path is returned; otherwise the function returns failure.

Algorithm 3 GENERATE_PATH

Require: $x_{init}, x_{end}, K, \rho, \Delta t, X, X_{obs}, d_{min}$
1: $V_{\tau_a} \leftarrow \{x_{init}\}$; $E_{\tau_a} \leftarrow \emptyset$;
2: $V_{\tau_b} \leftarrow \{x_{end}\}$; $E_{\tau_b} \leftarrow \emptyset$;
3: **for** $k = 1$ to K **do**
4: $x_{rand} \leftarrow$ RANDOM_STATE(X);
5: **if** EXTEND($\tau_a, x_{rand}, \rho, \Delta t, X_{obs}, d_{min}$) $\neq$ *Trapped* **then**
6: **if** EXTEND($\tau_b, x_{new}, \rho, \Delta t, X_{obs}, d_{min}$) $=$ *Reached* **then return** PATH(τ_a, τ_b);
7: **end if**
8: **end if**
9: SWAP(τ_a, τ_b);
10: **end for**
11: **return** *Failure*

Algorithm 3 is the bidirectional RRT algorithm presented in [13], where:

- x_{init} is the initial state.
- x_{end} is the ending state.
- $\tau_a = (V_{\tau_a}, E_{\tau_a})$ is an RRT starting at x_{init}.
- $\tau_b = (V_{\tau_b}, E_{\tau_b})$ is an RRT starting at x_{end}.
- PATH(τ_a, τ_b) is a function to compute a path from the initial node of τ_a to the initial node of τ_b. Both RRTs must be connected.
- SWAP(τ_a, τ_b) is a procedure to interchange the values of τ_a and τ_b.

The rest of variables have the same meaning than the variables used in Algorithms 1 and 2.

In this paper we propose the Algorithm 4 as a parallel version of the bidirectional RRT algorithm, where τ_a and τ_b are built at the same time.

Algorithm 4 GENERATE_PATH_PARALLEL

Require: $(x_{init}, x_{end}, K, \rho, \Delta t, X, X_{obs}, d_{min}$
1: $V_{\tau_a} \leftarrow \{x_{init}\}$; $E_{\tau_a} \leftarrow \emptyset$;
2: $V_{\tau_b} \leftarrow \{x_{end}\}$; $E_{\tau_b} \leftarrow \emptyset$;
3: **for** $k = 1$ to K **do**
4: $x_{rand,a} \leftarrow$ RANDOM_STATE(X);
5: $x_{rand,b} \leftarrow$ RANDOM_STATE(X);
6: **loop**
7: $result_a$ = EXTEND($\tau_a, x_{rand,a}, \rho, \Delta t, X_{obs}, d_{min}$);
8: $result_b$ = EXTEND($\tau_b, x_{rand,b}, \rho, \Delta t, X_{obs}, d_{min}$);
9: **end loop**
10: **if** $result_a \neq$ *Trapped* **then**
11: **if** EXTEND($\tau_b, x_{new}, \rho, \Delta t, X_{obs}, d_{min}$) = *Reached* **then return** PATH(τ_a, τ_b);
12: **end if**
13: **end if**
14: **if** $result_b \neq$ *Trapped* **then**
15: **if** EXTEND($\tau_a, x_{new}, \rho, \Delta t, X_{obs}, d_{min}$) = *Reached* **then return** PATH(τ_a, τ_b);
16: **end if**;
17: **end if**;
18: **end for**;
19: **return** *Failure*

4 Random enzymatic numerical P systems with proteins and shared memory

In this section a variant of *enzymatic numerical P systems* incorporating new features is presented, in order to simulate RRT algorithms.

A random enzymatic numerical P systems with proteins and shared memory (RENPSM, for short) of degree (p, q), $p, q \geq 1$ is a tuple $(H, \mu, P, E_{mem}, E_{mem}(0), \{P_h(0), Var_h, Var_h(0), Pr_h) \mid h \in H\}, \mathcal{R}, h_a, h_b)$, where:

1. $H = \{1, \ldots, p \cdot q\} \cup \{v, mem\}$, $mem \notin \{1, \ldots, p \cdot q\}$, $v \notin \{mem, 1, \ldots, p \cdot q\}$, is the set of labels of the system;

2. μ is a dynamic membrane structure (rooted tree) initially consisting of one skin membrane with label v including two inner membranes labelled respectively with $h_a \in \{1, \ldots, p \cdot q\}$ and $h_b \in \{1, \ldots, p \cdot q\}$, $h_a \neq h_b$, in such manner that along the computation only child membranes of h_a and h_b will be created with labels in $\{1, \ldots, p \cdot q\}$. In Figure 3, it is represented the initial membrane structure;

3. mem is the label of a distinguished component (the shared memory of the system);

4. P is a finite set of objects, called catalyzer proteins, and $P_h(0)$ is the protein initially associated with region labelled by h;

5. E_{mem} is a finite set of variables, called enzymes, disjoint with Var_{mem}, and $E_{mem}(0)$ is the initial values of the enzymes;

6. $Var_h, h \in H$, is a finite set of variables $x_{j,h}$ associated with region labelled by h (a membrane or the shared memory), its values are natural numbers and the value of $x_{j,i}$ at time $t \in \mathbf{N}$ is denoted by $x_{j,i}(t)$;

7. $Var_h(0)$ is a vector that represents the initial values for variables in Var_h;

8. $Pr_h, h \in H$, is a finite set of programs associated with region labelled by h, having the following syntactical format $F(x_{1,h}, \ldots, x_{k_F,h}) \xrightarrow{e(F);\alpha(F)} c_1|v_1, \ldots, c_{n_F}|v_{n_F}$, where:

   - $F(x_{1,h}, \ldots, x_{k_F,h})$ is a computable function (the production function), being $x_{1,h}, \ldots, x_{k_F,h} \in Var_h$;
   - $c_1|v_1, \ldots, c_{n_F}|v_{n_F}$ is the repartition protocol associated with the program, being $c_1, \ldots, c_{n_F}$ natural numbers specifying the proportion of the current production distributed to variables $v_1, \ldots, v_{n_F} \in Var_h \cup Var_{par(h)} \cup Var_{ch(h)}$, where $par(h)$ is the parent of h and $ch(h)$ is the set of child of h in μ;
   - $e(F) \in E_h$ is an enzyme and $\alpha(F) \in P$ is a protein, both of them associated with program F, if no enzyme or protein is used in a program then it will be omitted;

9. $\mathcal{R}$ is a finite set of rules of the following form:

   - *Protein evolution rules*: $[\,\alpha \to \alpha'\,]_h$, where $h \in H, \alpha \in P$ and $\alpha' \in P$.
   - *Writing-only communication rules between the shared memory and the membranes*

     $$(h\,,\, X_h\,/Y_{h,mem}\,,\, mem)^W_\alpha$$

     where $X_h \in Var_h, Y_{h,mem} \in Var_{mem}, \alpha \in P$ in such manner that there is, at most, one rule for each membrane $h \in \{1, \ldots, p \cdot q\}$. Variables $Y_{h,mem}, Y_{h',mem}$ should be different for two membranes h, h'.
   - *Reading-only communication rules between the shared memory and the membranes:*

     $$(h\,,\, X_h\,/Y_{mem}\,,\, mem)^R_\alpha$$

     where $X_h \in Var_h, Y_{mem} \in Var_{mem}, \alpha \in P$. Variable Y_{mem} is the same for each $h \in \{1, \ldots, p \cdot q\}$.
   - *Membrane creation rules*:

     $$\left[\ \left[\ X_{1,h}\,,\, X_{2,h},\, ,\, \ldots\, ,\, X_{n,h}\ \right]_h\ \right]_{h'}\,;\,\alpha$$

     where $h, h' \in \{1, \ldots p \cdot q \cdot r\}$ are different, $\alpha \in P$ and $X_{1,h}\,,\, \ldots\,,\, X_{n,h} \subseteq Var_h$.

The term *region* h ($h \in H$) is used to refer to membrane h in the case $h \in \{1, \ldots, p \cdot q\} \cup \{v\}$, as well as to refer to the shared memory in the case $h = mem$.
Next, we describe the semantics of RENPSHs. A *configuration* of a RENPSH at any instant t is described by the current membrane structure μ, together with proteins and all values of the variables and enzymes associated with all regions.
The *initial configuration* is $(\mu, E_{mem}(0), \{P_h(0), Var_h(0) | h \in H\})$, where $\mu = [[\,]_{h_a}[\,]_{h_b}]_v$. We will call μ_a (resp. μ_b) to the membrane structure rooted in membrane h_a (resp. h_b).

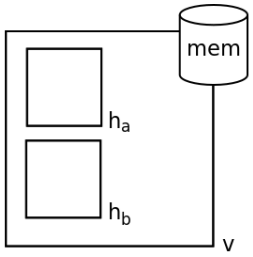


Figure 3: The initial membrane structure with a representation of the shared memory.

A program $F(x_{1,h},\ldots,x_{k_F,h}) \xrightarrow{e_F;p_F} c_1|v_1,\ldots,c_{n_F}|v_{n_F}$ associated with a region is applicable to a configuration $\mathcal{C}_t$ at moment t, if the value of $e(F)$ at that instant is greater than $min\{x_{1,h}(t),\ldots,x_{k_F,h}(t)\}$ and protein $\alpha(F)$ is inside the region h of $\mathcal{C}_t$. When applying such a program, variables associated with $\mathcal{C}_t$ are processed as follows: first, the value $F(x_{1,h}(t),\ldots,x_{k_F,h}(t))$ is computed as well as the value

$$q(t) = \frac{F(x_{1,h}(t),\ldots,x_{k_F,h}(t))}{c_1+\cdots+c_{n_F}}$$

This value represents the *unary portion* at instant t to be distributed among variables $v_1,\ldots,v_{n_F}$ according to the repartition expression. Thus, $q(t)\cdot c_s$ is the contribution added to the current value of v_s $(1\leq s\leq n_F)$, at step $t+1$. So, $v_s(t+1) = v_s(t) + q(t)\cdot c_s$ and $v_s(t)$ become zero, i.e, it is assumed that variable v_s is "consumed" when the production function is used and other variables retain their values. Each program in each membrane can only be used once in every computation step, and all the programs are executed in parallel.

A protein evolution rule $[\,\alpha\rightarrow\alpha'\,]_h$ is applicable to a configuration $\mathcal{C}_t$ at moment t if protein α is in membrane h of $\mathcal{C}_t$. When applying such a rule the protein α in h evolves to protein α' in h. These rules are applied in a maximal manner.

A writing-only communication rule between the shared memory and the membranes,

$$(h\,,\,X_h\,/Y_{h,mem}\,,\,mem)^W_\alpha$$

,

is applicable to a configuration $\mathcal{C}_t$ at moment t if protein α is in membrane h of $\mathcal{C}_t$. When applying such a rule the value $X_h(t)$ is assigned to the variable $Y_{h,mem}(t+1)$ of the shared memory, that is $Y_{h,mem}(t+1)\leftarrow X_h(t)$. These rules are applied in a maximal manner.

A reading-only communication rule between the shared memory and the membranes,

$$(h\,,\,X_h\,/Y_{mem}\,,\,mem)^R_\alpha$$

is applicable to a configuration $\mathcal{C}_t$ at moment t if protein α is in membrane h of $\mathcal{C}_t$. When applying such a rule the value $Y_{mem}(t)$ is assigned to the variable $X_h(t+1)$ of membrane h, that is $X_h(t+1)\leftarrow Y_{mem}(t)$. These rules are applied in a maximal manner.

A membrane creation rule

$[\,[X_{1,h}\,,\,\ldots\,,\,X_{n,h}\,]_h\,]_{h'}\,;\,\alpha$ is applicable to a configuration $\mathcal{C}_t$ at moment t if protein α is in membrane h' of $\mathcal{C}_t$. When applying such a rule, a new membrane labelled by h is created in such manner that h' is the parent of h and the set of its variables is $Var_h = \{X_{1,h}\,,\,\ldots\,,\,X_{n,h}\}$.

Given a random enzymatic numerical P system with proteins and shared memory Π, we say that configuration $\mathcal{C}_t$ at time t yields configuration $\mathcal{C}_{t+1}$ in one transition step if we can pass from $\mathcal{C}_t$ to $\mathcal{C}_{t+1}$ by applying in parallel each program in each membrane only once, and by applying the rules in a maximal parallel way following the previous remarks. A *computation* of Π is a (finite

or infinite) sequence of configurations such that: (a) the first term is the initial configuration of the system; (b) for each $n \geq 2$, the n-th configuration of the sequence is obtained from the previous configuration in one transition step; and (c) if the sequence is finite (called *halting computation*) then the last term is a *halting configuration* (a configuration where no rule of the system is applicable to it). All the computations start from an initial configuration and proceed as stated above; only halting computations give a result, which is encoded by the objects present in the output region i_{out} associated with the halting configuration. If $\mathcal{C} = \{\mathcal{C}_t\}_{t<r+1}$ of Π ($r \in \mathbb{N}$) is a halting computation, then the *length* of $\mathcal{C}$, denoted by $|\mathcal{C}|$, is r. For each i ($1 \leq i \leq q$), we denote by $\mathcal{C}_t(i)$ the finite multiset of objects over Γ contained in all membranes labelled by i at configuration $\mathcal{C}_t$.

5 Simulation of one iteration of the bidirectional RRT algorithm for path planning

The input of the bidirectional RRT algorithm generating a global path for a robot trajectory consists of the following parameters $(x_{init}, x_{end}, K, \rho, \Delta t, X, X_{obs}, d_{min})$, where:

- x_{init} is the initial state.
- x_{end} is the ending state.
- K is the number of iterations to find the path.
- Δt is a fixed amount of time for transitions.
- X is the state space.
- $X_{obs} \subseteq X$ is the obstacle state space.
- d_{min} is the minimum distance threshold according to some distance metric ρ in order to include a new node in the RRT.

For mobile robots in a 2D environment, the state space is given by (x, y, θ), i.e, the Cartesian coordinates (x, y) and the heading angle θ of all the possible robot poses. However, the angle θ has been omitted in this solution to reduce the size of the problem and the state space is given by (x, y) considering the Euclidean distance as distance metric. In this case, a holonomic robot can follow the RRT by performing motions in straight line, otherwise a nonholonomic robot can include rotations in-place. Moreover, any state or position $(i, j) \in \{0, \ldots p-1\} \times \{0, \ldots q-1\}$ in an Euclidean space constrained by a rectangle of p width and q height distance units can be encoded by the natural number $j \cdot p + i + 1$. In such a manner, given a natural number n encoding a state (i, j), the following holds: $i = rm(n-1, p)$ and $j = qt(n-1, p)$.

One iteration of the parallel bidirectional RRT algorithm defined in Algorithm 4 will be simulated by a RENPSM of degree (p, q)

$\Pi = (H, \mu, P, E_{mem}, E_{mem}(0),$
$\{(P_h(0), Var_h, Var_h(0), Pr_h) \mid h \in H\}, \mathcal{R}, h_a, h_b)$
defined as follows:

- $H = \{1, \ldots, p \cdot q\} \cup \{v, mem\}$, $v \notin \{1, \ldots, p \cdot q\}$, $mem \notin \{1, \ldots, p \cdot q\}$.
- $\mu = [[\,]_{h_a}[\,]_{h_b}]_v$ with $h_a \in \{1, \ldots, p \cdot q\}$, $h_b \in \{1, \ldots, p \cdot q\}$ and $h_a \neq h_b$. We call μ_a to the membrane structure rooted on h_a and μ_b to the one rooted on h_b.

– $P = \{\alpha_i \mid 1 \leq i \leq 18\}$, and $P_h(0) = \{\alpha_1\}$, for each $h \in H$.

– $E_{mem} = \{FlagA_{mem}, FlagB_{mem}, p \cdot q + 1\}$ and $E_{mem}(0) = \{p \cdot q + 1\}$.

– The set of variables is:

  – $Var_h = \{X_{1,h}, X_{2,h}, Y_{1,h}, Y_{2,h}, Z_{1,h}, Z_{2,h}, D_h\}$, for each $h, 1 \leq h \leq p \cdot q$.
  – $Var_{mem} = \{X_{1,mem}, X_{2,mem}, X_{3,mem}, X_{4,mem}\} \cup$
    $\{Y_{1,mem}, Y_{2,mem}, Y_{3,mem}, Y_{4,mem}\} \cup$
    $\{Z_{1,mem}, Z_{2,mem}, Z_{3,mem}, Z_{4,mem}\} \cup$
    $\{U_{1,mem}, U_{2,mem}, U_{3,mem}, U_{4,mem}\} \cup$
    $\{A_{mem}, B_{mem}, NA_{mem}, NB_{mem}, Halt_{mem}\} \cup$
    $\{A_{h,mem}, B_{h,mem} \mid 1 \leq h \leq p \cdot q\}$.
  – Initially, all variables in $Var_h (h \neq h_a \wedge h \neq h_b)$ and all variables in Var_{h_a} and Var_{h_b} different to $Y_{1,h}, Y_{2,h}$, are equal to zero. Besides, initially the tuple (Y_{1,h_a}, Y_{2,h_a}) (resp. (Y_{1,h_b}, Y_{2,h_b})) provides the position of the initial state of the robot h_a (resp. the position of the final state of the robot h_b).
  – If the value of variable $Halt_{mem}$ is equals to 1, then the computation stops.

- Next, the finite set of programs P_{r_h} and the set of rules $\mathcal{R}$ of the system are defined according to the requirements to simulate the bidirectional RRT algorithm.

- In order to synchronize the sequence of an iteration, for each $h \in H$ the protein evolution rules $[\,\alpha_i \to \alpha_{i+1}\,]_h$, for $1 \leq i \leq 17$, and $[\,\alpha_{18} \to \alpha_1\,]_h$ are considered.

- Four random numbers are generated in the shared memory:

$$\begin{cases} Production\ function : F(X_{1,mem}) = \\ Random(i, 0 \leq i < p) \\ Repartition\ protocol : 1|X_{1,mem} \\ Protein : \alpha_1 \end{cases}$$

$$\begin{cases} Production\ function : F(X_{2,mem}) = \\ Random(i, 0 \leq i < q) \\ Repartition\ protocol : 1|X_{2,mem} \\ Protein : \alpha_1 \end{cases}$$

$$\begin{cases} Production\ function : F(X_{3,mem}) = \\ Random(i, 1 \leq i < p) \\ Repartition\ protocol : 1|X_{3,mem} \\ Protein : \alpha_1 \end{cases}$$

$$\begin{cases} Production\ function : F(X_{4,mem}) = \\ Random(i, 1 \leq i < q) \\ Repartition\ protocol : 1|X_{4,mem} \\ Protein : \alpha_1 \end{cases}$$

- Each membrane $h \in \mu_a$ will read the random numbers $X_{1,mem}$, $X_{2,mem}$. Each membrane $h \in \mu_b$ will read the random numbers $X_{3,mem}$, $X_{4,mem}$.

$$\begin{cases} (h\,,\, X_{1,h} \,/\, X_{1,mem}\,,\, mem)^R_{\alpha_2} : h \in \mu_a \\ \\ (h\,,\, X_{2,h} \,/\, X_{2,mem}\,,\, mem)^R_{\alpha_2} : h \in \mu_a \end{cases}$$

$$\begin{cases} (h\,,\, X_{1,h} \,/\, X_{3,mem}\,,\, mem)^R_{\alpha_2} : h \in \mu_b \\ \\ (h\,,\, X_{2,h} \,/\, X_{4,mem}\,,\, mem)^R_{\alpha_2} : h \in \mu_b \end{cases}$$

- For each membrane $h \in \{\mu_a, \mu_b\}$, the distance D_h between its position $(Y_{1,h}, Y_{2,h})$ and the position given by the generated random numbers $(X_{1,h}, X_{2,h})$ is computed. For the remaining membranes, $D_h = p \cdot q + 1$.

$$\begin{cases} \textit{Production function} : F(X_{1,h}, X_{2,h}, Y_{1,h}, Y_{2,h}) = \\ \quad \begin{cases} \sqrt{\sum_{j=1}^{2}(X_{j,h} - Y_{j,h})^2} & \text{if} \quad h \in \{\mu_a, \mu_b\} \\ p \cdot q + 1 & \text{if} \quad h \notin \{\mu_a, \mu_b\} \end{cases} \\ \textit{Repartition protocol} : 1|D_h \\ \textit{Protein} : \alpha_3 \end{cases}$$

- Each membrane h writes its value D_h to the shared memory.

$$\begin{cases} (h\,,\, D_h \,/\, A_{h,mem}\,,\, mem)^W_{\alpha_4} : h \in \mu_a \\ \\ (h\,,\, D_h \,/\, B_{h,mem}\,,\, mem)^W_{\alpha_4} : h \in \mu_b \end{cases}$$

- The minimum of all distances $A_{h,mem}$ is computed in the shared memory.
  - *Production function*: $F(A_{1,mem}, \ldots, A_{p \cdot q,mem}) = \min\{A_{1,mem}, \ldots, A_{p \cdot q,mem}\}$
  - *Repartition protocol*: $1|A_{mem}$
  - *Protein*: α_5

- The minimum of all distances $B_{h,mem}$ is computed in the shared memory.
  - *Production function*: $F(B_{1,mem}, \ldots, B_{p \cdot q,mem}) = \min\{B_{1,mem}, \ldots, B_{p \cdot q,mem}\}$
  - *Repartition protocol*: $1|B_{mem}$
  - *Protein*: α_5

- Variable (enzyme) $FlagA_{mem}$ is set to zero if $A_{mem} \leq Threshold$.
  - *Production function*: $F(A_{mem}) = \begin{cases} 0 & \text{if } A_{mem} \leq Threshold \\ p \cdot q + 1 & \text{otherwise} \end{cases}$
  - *Repartition protocol*: $1|FlagA_{mem}$
  - *Protein*: α_6

- Variable (enzyme) $FlagB_{mem}$ is set to zero if $B_{mem} \leq Threshold$.

- *Production function*: $F(B_{,mem}) = \begin{cases} 0 & \text{if } B_{mem} \leq Threshold \\ p \cdot q + 1 & \text{otherwise} \end{cases}$
- *Repartition protocol*: $1|FlagB_{mem}$
- *Protein*: α_6

- The label $near_a$, corresponding to the closer membrane to the randomly generated point for μ_a, is obtained.
  - *Production function*: $F(A_{1,mem}, \ldots, A_{p \cdot q,mem}) = \text{arg-min}\{A_{1,mem}, \ldots, A_{p \cdot q,mem}\}$
  - *Repartition protocol*: $1|NA_{mem}$
  - *Protein*: α_7
  - *Enzyme*: $FlagA_{mem}$

- The label $near_b$, corresponding to the closer membrane to the randomly generated position for μ_b, is obtained.
  - *Production function*: $F(B_{1,mem}, \ldots, B_{p \cdot q,mem}) = \text{arg-min}\{B_{1,mem}, \ldots, B_{p \cdot q,mem}\}$
  - *Repartition protocol*: $1|NB_{mem}$
  - *Protein*: α_7
  - *Enzyme*: $FlagB_{mem}$

- The position of membrane $near_a$ is computed.

$$\begin{cases} Production\ function : F(NA_{mem}) = rm(NA_{mem} - 1, p) \\ Repartition\ protocol : 1|Y_{1,mem} \\ Protein : \alpha_8 \\ Enzyme : FlagA_{mem} \end{cases}$$

$$\begin{cases} Production\ function : F(NA_{mem}) = qt(NA_{mem} - 1, p) \\ Repartition\ protocol : 1|Y_{2,mem} \\ Protein : \alpha_8 \\ Enzyme : FlagA_{mem} \end{cases}$$

- The position of membrane $near_b$ is computed.

$$\begin{cases} Production\ function : F(NB_{mem}) = rm(NB_{mem} - 1, p) \\ Repartition\ protocol : 1|Y_{3,mem} \\ Protein : \alpha_8 \\ Enzyme : FlagB_{mem} \end{cases}$$

$$\begin{cases} Production\ function : F(NB_{mem}) = qt(NB_{mem} - 1, p) \\ Repartition\ protocol : 1|Y_{4,mem} \\ Protein : \alpha_8 \\ Enzyme : FlagB_{mem} \end{cases}$$

- The unitary vectors are created in the shared memory.

$$\begin{cases} Production\ function: \\ F(X_{1,mem}, X_{2,mem}, Y_{1,mem}, Y_{2,mem}) = \\ \frac{X_{1,mem}-Y_{1,mem}}{\sqrt{\sum_{j=1}^{2}(X_{j,mem}-Y_{j,mem})^2}} \\ Repartition\ protocol: 1|U_{1,mem} \\ Protein: \alpha_9 \\ Enzyme: FlagA_{mem} \end{cases}$$

$$\begin{cases} Production\ function: \\ F(X_{1,mem}, X_{2,mem}, Y_{1,mem}, Y_{2,mem}) = \\ \frac{X_{2,mem}-Y_{2,mem}}{\sqrt{\sum_{j=1}^{2}(X_{j,mem}-Y_{j,mem})^2}} \\ Repartition\ protocol: 1|U_{2,mem} \\ Protein: \alpha_9 \\ Enzyme: FlagA_{mem} \end{cases}$$

$$\begin{cases} Production\ function: \\ F(X_{3,mem}, X_{4,mem}, Y_{3,mem}, Y_{4,mem}) = \\ \frac{X_{3,mem}-Y_{3,mem}}{\sqrt{\sum_{j=3}^{4}(X_{j,mem}-Y_{j,mem})^2}} \\ Repartition\ protocol: 1|U_{3,mem} \\ Protein: \alpha_9 \\ Enzyme: FlagB_{mem} \end{cases}$$

$$\begin{cases} Production\ function: \\ F(X_{3,mem}, X_{4,mem}, Y_{3,mem}, Y_{4,mem}) = \\ \frac{X_{4,mem}-Y_{4,mem}}{\sqrt{\sum_{j=3}^{4}(X_{j,mem}-Y_{j,mem})^2}} \\ Repartition\ protocol: 1|U_{4,mem} \\ Protein: \alpha_9 \\ Enzyme: FlagB_{mem} \end{cases}$$

- Variable (enzyme) $FlagA_{mem}$ is set to zero if there is collision for μ_a.
  - *Production function*: $F(Y_{1,mem}, Y_{2,mem}, U_{1,mem}, U_{2,mem}) =$
    $\begin{cases} 0 & \text{if } COLLISION(Y_{1,mem}, Y_{2,mem}, \\ U_{1,mem}, U_{2,mem}) & \\ p \cdot q + 1 & \text{otherwise} \end{cases}$
  - *Repartition protocol*: $1|FlagA_{mem}$
  - *Protein*: α_{10}
  - $COLLISION$ is a function returning *true* if there are static obstacles in a linear trajectory starting at $(Y_{1,mem}, Y_{2,mem})$ and applying a motion $(U_{1,mem}, U_{2,mem})$ for Δt time.
- Variable (enzyme) $FlagB_{mem}$ is set to zero if there is collision for μ_b.
  - *Production function*: $F(Y_{3,mem}, Y_{4,mem}, U_{3,mem}, U_{4,mem}) =$
    $\begin{cases} 0 & \text{if } COLLISION(Y_{3,mem}, Y_{4,mem}, \\ U_{3,mem}, U_{4,mem}) & \\ p \cdot q + 1 & \text{otherwise} \end{cases}$

- *Repartition protocol*: $1|FlagB_{mem}$
- *Protein*: α_{10}
- $COLLISION$ is a function returning *true* if there are static obstacles in a linear trajectory starting at $(Y_{3,mem}, Y_{4,mem})$ and applying a motion $(U_{3,mem}, U_{4,mem})$ for Δt time.

- Positions of new membranes are computed in the shared memory.

$$\begin{cases} Production\ function: \\ F(Y_{1,mem}, U_{1,mem}) = round(Y_{1,mem} + U_{1,mem} \cdot \Delta t) \\ Repartition\ protocol: 1|Z_{1,mem} \\ Protein: \alpha_{11} \\ Enzyme: FlagA_{mem} \end{cases}$$

$$\begin{cases} Production\ function: \\ F(Y_{2,mem}, U_{2,mem}) = round(Y_{2,mem} + U_{2,mem} \cdot \Delta t) \\ Repartition\ protocol: 1|Z_{2,mem} \\ Protein: \alpha_{11} \\ Enzyme: FlagA_{mem} \end{cases}$$

$$\begin{cases} Production\ function: \\ F(Y_{3,mem}, U_{3,mem}) = round(Y_{3,mem} + U_{3,mem} \cdot \Delta t) \\ Repartition\ protocol: 1|Z_{3,mem} \\ Protein: \alpha_{11} \\ Enzyme: FlagB_{mem} \end{cases}$$

$$\begin{cases} Production\ function: \\ F(Y_{4,mem}, U_{4,mem}) = round(Y_{4,mem} + U_{4,mem} \cdot \Delta t) \\ Repartition\ protocol: 1|Z_{4,mem} \\ Protein: \alpha_{11} \\ Enzyme: FlagB_{mem} \end{cases}$$

- The membranes labelled by NA_{mem} and NB_{mem} will read the positions corresponding to the new membranes from the shared memory.

$$\begin{cases} (NA_{mem}\ ,\ Z_{1,NA_{mem}}\ /\ Z_{1,mem}\ ,\ mem)^R_{\alpha_{12}} \\ \\ (NA_{mem}\ ,\ Z_{2,NA_{mem}}\ /\ Z_{2,mem}\ ,\ mem)^R_{\alpha_{12}} \end{cases}$$

$$\begin{cases} (NB_{mem}\ ,\ Z_{1,NB_{mem}}\ /\ Z_{3,mem}\ ,\ mem)^R_{\alpha_{12}} \\ \\ (NB_{mem}\ ,\ Z_{2,NB_{mem}}\ /\ Z_{4,mem}\ ,\ mem)^R_{\alpha_{12}} \end{cases}$$

- A child membrane with position $(Z_{1,NA_{mem}}, Z_{2,NA_{mem}})$ is created in μ_a.

$$\left[\left[\begin{array}{cc} X_{1,h} & X_{2,h} \\ Y_{1,h} & Y_{2,h} \\ Z_{1,h} & Z_{2,h} \\ D_h & \end{array} \right]_h \right]_{NA_{mem}}$$

Being $h = Z_{2,NA_{mem}} \cdot p + Z_{1,NA_{mem}} + 1$.
This rule is mediated by protein α_{13}.

- A child membrane with position $(Z_{1,NB_{mem}}, Z_{2,NB_{mem}})$ is created in μ_b.

$$\left[\left[\begin{array}{cc} X_{1,h} & X_{2,h} \\ Y_{1,h} & Y_{2,h} \\ Z_{1,h} & Z_{2,h} \\ D_h & \end{array} \right]_h \right]_{NB_{mem}}$$

  Being $h = Z_{2,NB_{mem}} \cdot p + Z_{1,NB_{mem}} + 1$.
  This rule is mediated by protein α_{13}.

- Each membrane in μ_a reads the position of the new membrane created in μ_b

$$\begin{cases} (h\,,\, X_{1,h} \,/\, Z_{3,mem}\,,\, mem)^R_{\alpha_{14}} : h \in \mu_a \\ \\ (h\,,\, X_{2,h} \,/\, Z_{4,mem}\,,\, mem)^R_{\alpha_{14}} : h \in \mu_a \end{cases}$$

- Each membrane in μ_b reads the pose of the new membrane created in μ_a

$$\begin{cases} (h\,,\, X_{1,h} \,/\, Z_{1,mem}\,,\, mem)^R_{\alpha_{14}} : h \in \mu_b \\ \\ (h\,,\, X_{2,h} \,/\, Z_{2,mem}\,,\, mem)^R_{\alpha_{14}} : h \in \mu_b \end{cases}$$

- For each membrane $h \in \{\mu_a, \mu_b\}$, the distance D_h between its position $(Y_{1,h}, Y_{2,h})$ and the position given by the new membrane in the other membrane structure is computed. For the remaining membranes, $D_h = p \cdot q + 1$.

$$\begin{cases} \textit{Production function} : F(X_{1,h}, X_{2,h}, Y_{1,h}, Y_{2,h}) = \\ \quad \begin{cases} \sqrt{\sum_{j=1}^{2}(X_{j,h} - Y_{j,h})^2} & \text{if} \quad h \in \{\mu_a, \mu_b\} \\ p \cdot q + 1 & \text{if} \quad h \notin \{\mu_a, \mu_b\} \end{cases} \\ \textit{Repartition protocol} : 1|D_h \\ \textit{Protein} : \alpha_{15} \end{cases}$$

- Each membrane h writes its value D_h to the shared memory.

$$\begin{cases} (h\,,\, D_h \,/\, A_{h,mem}\,,\, mem)^W_{\alpha_{16}} : h \in \mu_a \\ \\ (h\,,\, D_h \,/\, B_{h,mem}\,,\, mem)^W_{\alpha_{16}} : h \in \mu_b \end{cases}$$

- The minimum of all distances $A_{h,mem}$ is computed in the shared memory.
  - *Production function*: $F(A_{1,mem}, \ldots, A_{p \cdot q,mem}) = \min\{A_{1,mem}, \ldots, A_{p \cdot q,mem}\}$
  - *Repartition protocol*: $1|A_{mem}$
  - *Protein*: α_{17}

- The minimum of all distances $B_{h,mem}$ is computed in the shared memory.
  - *Production function*: $F(B_{1,mem}, \ldots, B_{p \cdot q,mem}) = \min\{B_{1,mem}, \ldots, B_{p \cdot q,mem}\}$

  - *Repartition protocol*: $1|B_{mem}$
  - *Protein*: α_{17}

- If $A_{mem} \leq d_{min}$ or $B_{mem} \leq d_{min}$ then the RRTs have been connected and the computation must halt.

  - *Production function*: $F(A_{mem}, B_{mem}) =$
    $\begin{cases} 1 & \text{if } A_{mem} \leq Threshold \vee B_{mem} \leq Threshold \\ 0 & \text{otherwise} \end{cases}$
  - *Repartition protocol*: $1|Halt_{mem}$
  - *Protein*: α_{18}

6 A simulator based on C++ and ROS

A C++ simulator has been developed within the ROS [30] framework. It can be downloaded from `https://github.com/RGNC/renpsm`. The experiments have been conducted by using a dual-wheeled nonholonomic robot (the Pioneer 3-DX) in two virtual environments. The software is composed by three modules:

- MobileSim module [31]: It receives the static information about the map, as well as motion commands (v, ω) and generates the wheels odometry and information related to sensors (laser rangefinder for obstacle detection). It moves the simulated robot in the virtual environment.

- RENPSM module: It receives the information about the map, as well as the information about odometry and sensors and the goal of the robot. It computes a bidirectional RRT by using a RENPSM simulator and finally it sends a sequence of motion commands to the MobileSim module.

- RVIZ module [32]: This module is used for visualization. It receives the static information about the map, as well as all the information generated by the MobileSim module and several visual markers generated by the RENPSM module. It shows to the user all the information in real-time by using a 3D representation of the environment and the robot.

We have used two virtual environments, Figures 4 and 5 show the corresponding RVIZ visualization for each one before starting the robot motion, i.e, after generating the bidirection RRT by using the RENPSM module. Figure 6 is a second simulation for environment 2.

The first environment has been used for experimental validation of the RENPSM model by generating several simulations and comparing the resulting RRT visualizations with the ones generated with a conventional RRT software.

The second environment has been used for benchmarking, generating 1435 simulations by fixing the starting point and the goal of the robot and measuring the cost in distance of the generated path. The results are shown in Table 1.

We have measured the cost of an optimal path generated by hand (about $10m$) and, as expected, the cost of the best path generated by the bidirectional RRT is larger than the optimal cost, since the algorithm generates the first feasible path that can be found.

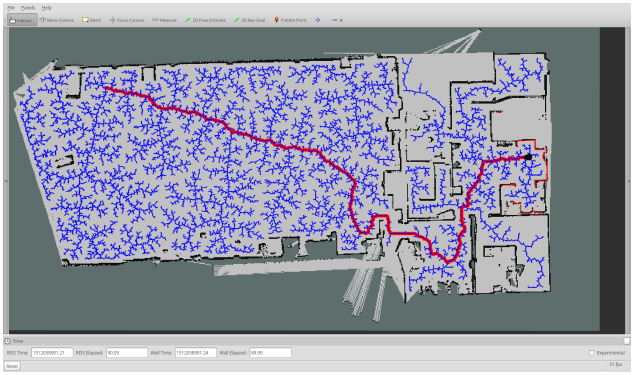


Figure 4: RVIZ visualization of a simulation in environment 1

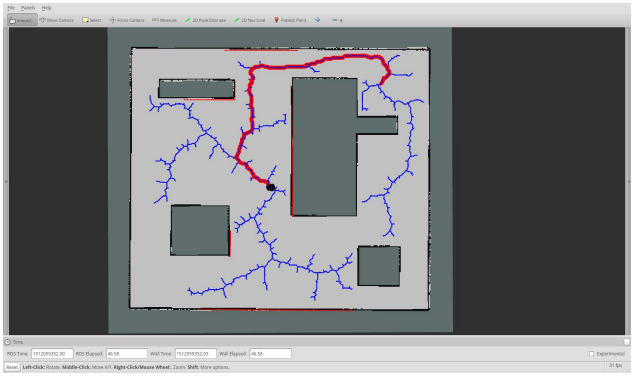


Figure 5: RVIZ visualization of a simulation in environment 2

7 Generation of ad-hoc RENPSM simulators with P-Lingua

P-Lingua is a language to define membrane computing models [3, 7, 14, 33], allowing to write definitions in a friendly way, as its syntax is close to standard scientific notation. In this section, we present an extension of P-Lingua for RENPSM. A parser based on Flex and Bison [35, 36] has been implemented to generate ad-hoc simulators for the defined models by using automatic programming. The source code of the simulators is generated in C language and OpenMP [37], providing multi-threading simulators for multicore processors. Finally, some experimental results are presented.

7.1 Extension of P-Lingua for RENPSM

New syntactic ingredients have been included to define models based RENPSM:

- The first line of a P-Lingua file defining a RENPSM model must be `@model<renpsm>`

Table 1: Benchmarking results

Min. cost	11.77 m
Max. cost	17.96 m
Average cost	13.42 m
Standard deviation	0.795 m
Experiments	1435

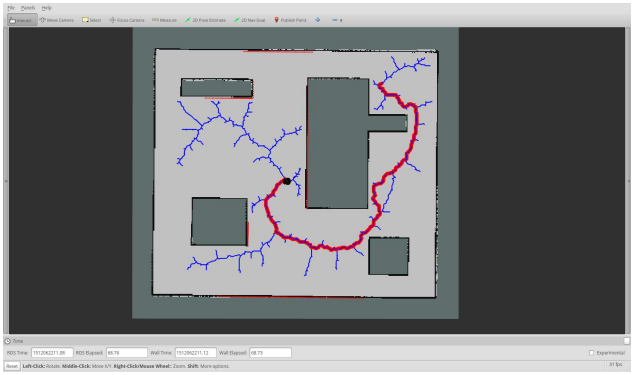


Figure 6: A second simulation in environment 2

- The initial membrane structure is defined as a two level cell-like membrane structure:

$$@\texttt{mu} = [[]'\texttt{h1}[]'\texttt{h2}...[]'\texttt{hN}]'\texttt{v}$$

  where `h1, h2,...,hN` and `v` are numeric labels in $\mathbb{N}$. The label 0 is used the shared memory. P-Lingua variables can optionally be used instead of literals.

- Each RENPSM variable includes at least one index. The last index of each variable is the label of the compartment containing the variable. For instance, the variable $X_{1,mem}$ is written `X{1,0}`. For the sake of simplicity, indexes will be omitted in the rest of this section.

- Proteins are written as common objects in P-Lingua.

- Production rules are written in the next manner:

$$\texttt{X} < -\texttt{func}(\texttt{param1}, \texttt{param2}, ..., \texttt{paramN}), \texttt{protein}?\texttt{enzyme}$$

  where `X` is a RENPSM variable; `func` is a production function with parameters `param1`, `param2`, ..., `paramN`, `protein` is an optional protein related to the rule and `enzyme` is an optional enzyme. There are a fixed set of production functions that can be used. In this paper, we have implemented the functions for the model in Section 5. More functions could be implemented in C language. The rule will be executed if the protein is present and the enzyme has a value greater than zero.

- Reading-only and writing-only communication rules are respectively written as follows:

$$\texttt{X} < -\texttt{Y}, \texttt{protein}?\texttt{enzyme}$$

$$\texttt{Y} < -\texttt{X}, \texttt{protein}?\texttt{enzyme}$$

  where `X` is a variable in the membrane structure, `Y` is a variable in the shared memory, `protein` is an optional protein related to the rule and `enzyme` is an optional enzyme. The rule will be executed if the protein is present and the enzyme has a value greater than zero.

- Membrane creation rules are written as follows:

$$[[]'\texttt{h1}]'\texttt{h2}, \texttt{protein}?\texttt{enzyme}$$

where `h1`, `h2` are labels that can be given by numeric literals, P-Lingua variables or even RENPSM variables; `protein` is an optional protein related to the rule and `enzyme` is an optional enzyme. The rule will be executed if the protein is present and the enzyme has a value greater than zero.

- Protein evolution rules are written in the next manner:

$$[\texttt{p1} < -\texttt{p2}]'\texttt{h}$$

where `p1 and p2` are proteins and `h` is the label of a compartment, given by a numeric literal or a P-Lingua variable.

- We have created an special iterator `x in H` to iterate all the membrane labels in the tree structure rooted at `H` (with a depth-first order).

- Other ingredients of P-Lingua, such as modules, indexes, iterators and comments can be used as usual in this extension.

The whole P-Lingua file defining the model presented in 5 can be downloaded from `https://github.com/RGNC/renpsm_openmp/blob/master/birrt_renpsm_test1.pli`.

7.2 A parser to generate ad-hoc RENPSM simulators

A parser based on Flex and Bison [35,36] has been implemented and can be downloaded from `https://github.com/RGNC/renpsm_openmp`.

The website includes instructions for compiling and running the experiments in this paper. The parser reads a P-Lingua file and generates a command-line ad-hoc simulator tool that accepts a PGM file defining the obstacles and generates another PGM file printing the generated RRT over the obstacles image. The source code of the simulator tool is generated in C with OpenMP, the command-line syntax for the generated simulator is:

```
./simulator [-t threads] [-s steps] [-d] [-r seed]
[-m obstacles.pgm] [-o output.pgm]
```

Where:

- `-t threads` is the number of threads to be used. Default is 4. If 1 thread is set, the simulator will be sequential.

- `-s steps` is the maximum number of computational steps to simulate. The simulator stops if the variable $Halt_{mem}$ is set to 1 or the number of steps is reached. Default is 1048576 steps.

- If `-d` is set, debug information will be prompted.

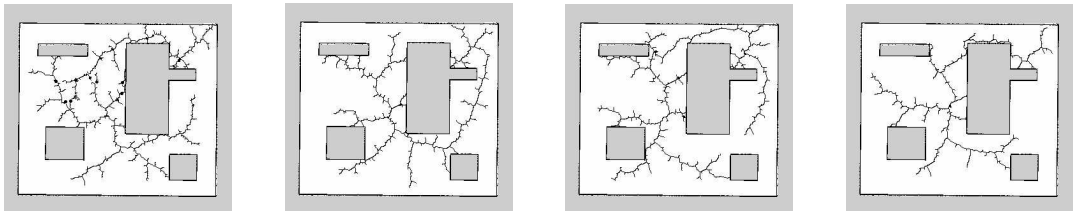
- `-r seed` defines the pseudo-random number generator seed. If no seed is configured, an arbitrary seed based on the current clock time will be used.

- `-m obstacles.pgm` is the PGM file defining the obstacle grid for the collision function.

- `-o output.pgm` is the PGM file to print the output RRT over the obstacle image.

(a) (b) (c) (d)

Figure 7: Example outputs for environment 1: (a) Example output 1; (b) Example output 2; (c) Example output 3; (d) Example output 4



(a) (b) (c) (d)

Figure 8: Example outputs for environment 2: (a) Example output 1; (b) Example output 2; (c) Example output 3; (d) Example output 4

7.3 Experimental results

We have used the same two environments presented in Section 6. The environment 1 is defined with the files: `birrt_renpsm_test2.pli`; `office.pgm`.

The environment 2 is defined with the files: `birrt_renpsm_test1.pli`; `map.pgm`.

Each environment uses a PGM file in gray scale for obstacle definition, where each pixel represents a region of 5 cm^2 in the real world. Each environment also uses a P-Lingua file with identical P system definition but different initial parameters p, q, x_0, y_0, x_1, y_1. The problem is to find a RRT representation to navigate from (x_0, y_0) to (x_1, y_1) in the environment of p pixels length and q pixels height given by the corresponding PGM file.

In Figures 7 and 8 there are four example outputs for each environment using the corresponding generated ad-hoc simulator with 8 threads and arbitrary pseudo-random number generator seeds based on the CPU clock time.

8 Conclusions

This paper deals with an algorithm belonging to a family widely used to solve the problem of motion planning in robots, e.g., the RRT algorithms. Such class of algorithms are based on the randomized exploration of the configuration space. This paper is an extension of [21]. In such a work, a variant of Enzymatic Numerical P systems, called *random enzymatic numerical*

P systems with proteins and shared memory (RENPSM, for short) was introduced. Besides, a simplified version of the standard bidirectional RRT algorithm was described by a RENPSM system capturing the semantics of the new variant, where maximal parallelism is used.

The main contribution of this paper with respect to [21] is to provide a novel approximation for software simulation by using automatic programming. We have implemented a tool for parsing P-Lingua files defining RENPSM models and generating source code in C and OpenMP for ad-hoc simulators. Thus, we have a flexible way to debug since we are using a language to define the models instead of hard-coding them in the source code. Moreover, the generated source code is able to run on multicore processors by using OpenMP.

Three main challenges are planned as future work. First, to provide a formal verification of such RENPSM systems, in the sense that they in fact simulate the RRT generation algorithm. The second challenge is to move to the RRT* algorithm [9], a variant of the initial algorithm that is able to approximate optimal motion planning with enough iterations. Finally, to provide real-life robot path planning experiments, by using a nonholonomic robot with kinodynamic and environment constraints.

We also propose to apply the simulation techniques introduced in this paper to other types of P systems walking towards a more generic software tool based on P-Lingua and automatic programming for generation of optimized ad-hoc simulators.

Acknowledgements

This work was supported by National Natural Science Foundation of China (61672437 and 61702428) and by Sichuan Science and Technology Program (2018GZ0086, 2018GZ0185).

Authors from the University of Seville also acknowledge the support of the research project TIN2017-89842-P, co-financed by *Ministerio de Economía, Industria y Competitividad (MINECO)* of Spain, through the *Agencia Estatal de Investigación (AEI)*, and by *Fondo Europeo de Desarrollo Regional (FEDER)* of the European Union.

Bibliography

[1] Astrom, K.J.; Hagglund, T. (1995). *PID Controllers: Theory, Design, and Tuning*, 1995

[2] Colomer, M.A.; Margalida, A.; D. Sanuy,D.; Pérez-Jiménez, M.J. (2011). A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study. *Ecological Modelling*, 222 (1), 33-47, 2011.

[3] Díaz-Pernil, D.; Pérez-Hurtado, I.; Pérez-Jiménez, M.J. ; Riscos-Núñez, A. (2009). A P-Lingua Programming Environment for Membrane Computing. *Lecture Notes in Computer Science*, 5391, 187–203, 2009.

[4] Coulter, C. (1992). *Implementation of the Pure Pursuit Path Tracking Algorithm*, Tech. Report, CMU-RI-TR-92-01, Robotics Institute, Carnegie Mellon University, 1992.

[5] Fox, D.; Burgard, W.; Thrun, S. (1997). The dynamic window approach to collision avoidance, *Robotics and Automation Magazine*, 4 (1), 23-33, 1997.

[6] Gao, Y.; Wu, X.; Liu, Y.; Li, J.M.; Liu J.H.(2017). A Rapid Recognition of Impassable Terrain for Mobile Robots with Low Cost Range Finder Based on Hypotheses Testing Theory, *International Journal of Computers Communications & Control*, 12(6), 813-823, 2017.

[7] García-Quismondo, M.; Gutiérrez-Escudero, R.; Martínez-del-Amor, M.A.; Orejuela-Pinedo, E.; Pérez-Hurtado, I. (2009). P-Lingua 2.0: A software framework for cell-like P systems. *International Journal of Computers Communications & Control*, 4(3), 234-243, 2009.

[8] Huang, S.; Dissanayake, G. (2016). *Robot Localization: An Introduction*, Wiley Encyclopedia of Electrical and Electronics Engineering, 2016

[9] Karaman, S.; Frazzoli, E. (2010). Incremental Sampling-based Algorithms for Optimal Motion Planning, *Robotics Science and Systems VI*, 1-9, 2010

[10] Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots, *Int J Robot Res*, 5(1), 90-98, 1986.

[11] Latombe, J.C. (1991). *Robot Motion Planning*, Kluwer Academic Publishers, Boston, MA, 1991.

[12] LaValle, S.M. (1998). *Rapidly-Exploring Random Trees: A New Tool for Path Planning*, Computer Science Dept., Iowa State University, October 1998.

[13] LaValle, S.M. ; Kuffner, J.J. (1999). Randomized kinodynamic planning, *Proceedings IEEE International Conference on Robotics and Automation*, 473-479, 1999.

[14] Martínez-del-Amor, M.A.; Pérez-Hurtado, I.; Pérez-Jiménez, M.J.; Riscos-Núñez, A. (2010). A P-Lingua based simulator for Tissue P Systems, *Journal of Logic and Algebraic Programming*, 79 (6), 374-382, 2010.

[15] Nash, A.; K. Daniel, Koenig,S.; Felner, A. (2010). Theta: Any-Angle Path Planning on Grids, *Journal of Artificial Intelligence Research*, 39, 533-579, 2010.

[16] Păun, G. (2010). Computing with membranes, *Journal of Computer and System Sciences*, 61 (1), 108-143, 2000.

[17] Păun, G., Păun R. (2006). Membrane Computing and Economics: Numerical P Systems, *Fundamenta Informaticae*, 73(1,2), 213–227, 2006.

[18] Pavel, A.; Arsene, O.; Buiu, C. (2010). Enzymatic Numerical P Systems - A New Class of Membrane Computing Systems, *Proceedings of IEEE fifth international conferenced on bio-inspired computing: Theories and applications (BIC-TA)*, 1331-1336, 2010.

[19] Pavel, A.; Vasile, C.; Dumitrache, I. (2012). Robot localization implemented with enzymatic numerical P systems, *Proceedings of the international conference on biomimetic and biohybrid systems*, 204-215, 2012.

[20] Pavel, A.; Buiu, C. (2012). Using enzymatic numerical P systems for modeling mobile robot controllers, *Natural Computing*, 11(3), 387-393, 2012.

[21] Pérez, I.; Pérez-Jiménez, M.J.; Zhang, G.; Orellana-Martín D. (2018). Robot path planning using rapidly-exporing random trees: A membrane computing approach, *2018 IEEE 7th International Conference on Computers Communications and Control, Proc. of* (ICCCC2018), Oradea, Romania, May 08-12, 37-46, 2018.

[22] Pérez-Jiménez, M.J.(2014); The P versus NP problem from Membrane Computing view, *European Review*, 22 (1), 18–33, 2014.

[23] Pérez-Hurtado, I. Pérez-Jiménez, M.J. (2017). Generation of rapidly-exploring random tree by using a new class of membrane systems. *Pre-proceedings of Asian Conference on Membrane Computing* (ACMC2017), Chengdu, China, September 21-25, 534-546, 2017.

[24] Romero-Campero, F.J.; Pérez-Jiménez, M.J. (2008). A Model of the Quorum Sensing System in Vibrio fischeri Using P Systems. *Artificial Life*, 14 (1), 95-109, 2008.

[25] Stentz, A. (1995). The Focussed D* Algorithm for Real-time Replanning, *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 2, 1652-1659, 1995.

[26] Wang, H.; Yu, Y.; Q. Yuan, Q. (2011). Application of Dijkstra algorithm in robot path-planning, *Proceedings of the 2nd International Conference on Mechanic Automation and Control Engineering*, 1067-1069, 2011.

[27] Wang, T.; Zhang, G.; Zhao, J.; He, Z.; Wang, J.; Pérez-Jiménez, M.J. (2015). Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural P systems. *IEEE Transactions on Power Systems*, 30(3), 1182 - 1194, 2015.

[28] Widyotriatmo, A.; Joelianto, E.; Prasdianto, A.; Bahtiar, H.; Nazaruddin, Y.Y. (2017). Implementation of Leader-Follower Formation Control of a Team of Nonholonomic Mobile Robots, *International Journal of Computers Communications & Control*, 12(6), 871-885, 2017.

[29] Zhang, G.; Perez-Jimenez, M.J.; Gheorghe, M. (2017). *Real-life Applications with Membrane Computing*, Series: Emergence, Complexity and Computation, Volume 25. Springer International Publishing, 2017.

[30] `http://www.ros.org`

[31] `http://www.mobilerobots.com/Software/MobileSim.aspx`

[32] `http://wiki.ros.org/rviz`

[33] `http://www.p-lingua.org/wiki/index.php/Main_Page`

[34] `https://developer.nvidia.com/cuda-zone`

[35] `https://github.com/westes/flexl`

[36] `https://www.gnu.org/software/bison/`

[37] `https://www.openmp.org/`