Cytos: Morphogenetic (M) Systems for Modeling and Experimentation

Vladimír Smolka[2], Jan Drastík[2], Max Garzon[1], and Petr Sosík[2]

[1] Computer Science, The University of Memphis, Tennessee, USA
[2] Research Institute of the IT4Innovations Centre of Excellence, Faculty of Philosophy and Science, Silesian University, Opava, Czech Republic

Abstract. Morphogenetic system (M system) is an abstract model motivated by key aspects of morphogenetic phenomena such as self-assembly, growth, homeostasis, self-reproduction and self-healing of evolving systems. Its original purpose is the study of these phenomena, both theoretically and by experimentation, at a computational level abstracted from a biological implementation. Application in biological modeling and research is also worth considering.
Mathematically, M systems rely on basic principles of membrane computing and self-assembly, as well as explicit emphasis on geometrical structures (location and shape) in 3D (or generally, dD) Euclidean space. Theoretical studies have shown that M systems are computational universal, as well as efficient in solving difficult NP problems. Moreover, they have also shown robustness to injuries and self-healing capabilities through extensive computer simulations of specific M systems modeling self-reproduction of a model of a basic eukaryotic cell.
As computer simulations play a crucial tool in study and applications of morphogenetics, we have developed a software package to implement M systems *in silico*. It consists of two modules, a simulation engine and a visualizing tool based on the Unity game engine. Due to the key role of geometry and self-assembly, we were unable to use known P systems modelling libraries, such as the P-Lingua. In this paper we present examples of its functionality and range of applications, and compare it with other simulators used in research of P systems.

1 Introduction

A morphogenetic (M) system is a formal model intended to address the general question of whether morphogenesis and its characteristic properties (such as internal dynamical homeostasis, self-reproduction, and self-healing) can be meaningfully understood from the perspective of information processing, at some level of abstraction. The potential importance of morphogenesis for information sciences was recognized already by Alan Turing in [14], by focusing on a possible mechanisms underlying memory pattern formation and their resilience in biological organisms.

Among recent attempts to address these questions we mention membrane computing [9] and virtual cells [13]. However, these models assume cell as an

atomic assembly unit of an abstract nature, while M systems construct a developmental structure from 1D or 2D primitives allowing for gradual self-assembly of 3D (multi-)cell-like forms. The fundamental ingredient of P systems is the membrane that separates a virtual cell from the external world or the various parts within it. M systems enrich this concept with spatial relationships and constraints in the organization of biological systems, including the role of *geometric shape* and *form*. A survey of results in membrane computing specifically related to morphogenesis ca nbe found in the introductory section of [11].

To implement space and geometry in M systems, we have employed and generalized another bio-inspired approach – abstract tile assembly, originated in DNA computing [5,15]. The resulting model of M systems demonstrated its computational universality (in the Turing sense), and also its computational efficiency in solving NP-hard problems in probabilistic polynomial time. Furthermore, its resistance to injuries and its self-healing capability has been theoretically confirmed in [11]. On the other hand, computational experiments [12] [11] have proven the ability of M systems to reflect, to some degree, the corresponding macroscopic observables of growth of biological organisms (independently of whether they faithfully describe factual atomic processes in biological organisms), while maintaining their computational feasibility. To carry out these experiments, we have implemented a software environment capable of simulating M systems. The environment was named Cytos, as the first experiments aimed at simulation of the cytoskeleton growth and its regulatory role in cell fission.

In this paper we describe the implementation and use of the Cytos package which consists of a simulation engine and a visualization tool powered by the Unity game engine. In Section 2 we summarize basic design principles and the structure of M systems. Section 3 introduces our simulation engine Cytos and examples of its use, with input in various formats. Section 3.5 focuses on the visualization tool, while Section 4 contains visualized examples of simulation of three M systems. A comparison with some other simulation packages (such as the P-lingua and MeCoSim) used in P systems research is made in Section 5. Finally, in Sec. 6, we discuss the results and present some open questions concerning further improvements of Cytos.

2 Morphogenetic systems

This section summarizes the definition of an M system and their basic properties. The reader is referred to [11,12] for a complete specification and other examples.

M systems partially rely on the concept of P systems with "proteins" on membranes [8] but, to distinguish the latter from biological proteins, the analogous term "protion" was is used in the framework of M systems. Thanks to the extension with explicit geometric features and self-assembly, an M system unfolds in a dD Euclidean space $\mathbb{R}^d$ in *discrete time steps*, although here we will assume 3D space throughout. There are three types of objects present in the system: *protions*, *tiles* and *floating objects.*

Floating objects are small shapeless atomic objects floating freely within the environment, although they have nonzero volume and a certain specific position in space at every moment. They can be carried through protion channels and participate in mutual reactions with other types of objects.

Tiles have a predefined size and shape, together with a specified position and orientation in space at every time. Tiles can stick together (assemble) along their edges or at selected points called *connectors* and covered with *glues.* Their connection is controlled by a pre-defined *glue relation* and connecting angle.

Protions are placed on tiles and, apart from acting as protion channels letting floating objects pass through, they can also catalyze their reactions.

Unlike typical membrane systems, *membranes* are not present at the outset. even implicitly. However, they can be self-assembled from tiles and more atomic components during the morphogenesis of the M system. Connected tiles can be also become disconnected and/or be destroyed under certain conditions specified by the rules defining their interactions in the M system.

2.1 Polytopic tiling

A basic element of the tiling is a d-dimensional tile shaped as a bounded convex polytope (d-polytope) [16], with faces of dimension $d-1$, called *facets*, separating them from the exterior. A polytope is the convex hull of an ordered list of its points extreme points in $\mathbb{R}^d$, called its vertices. Formally, an *m-dimensional tile* is defined as

$$t = (\Delta, \{c_1, \ldots, c_k\}, g_s), \text{ for } k \geq 0, \text{ where}$$

Δ is a bounded convex m-polytope,
$c_1, \ldots, c_k$ are its connectors,
$g_s \in G$ is the *surface glue*, where G is a finite set of *glues.*

Connectors define possible attachments of the tile to other tiles. Informally, a *connector* is a site on the surface of a tile specified by its shape, glue and connecting angle. A connector may be shaped as a point, a segment or a polygon. Generally, two connectors on neighboring tiles can connect together if they have identical shapes and their glues match in the glue relation defined bellow.

Definition 1. *A* polytopic tile system *in* $\mathbb{R}^d$ *is a construct* $T = (Q, G, \gamma, d_g, S)$, *where*

Q *is the set of tiles of dimensions* $\leq d$;
G *is the set of glues;*
$\gamma \subseteq G \times G$ *is the glue relation;*
$d_g \in \mathbb{R}_0^+$ *is the* gluing radius *(assumed to be small compared to tile sizes);*
S *is a finite multiset of seed tiles from* Q *randomly distributed in space.*

2.2 M system

Growth in an M system is determined by the geometrical structure of its underlying polytopic tile system. Unlike ordinary tiling assembly systems, new tiles attaching to an existing structure are not present in an arbitrary quantity, but can only be created by the application of rules of the M system to more elementary objects available in its environment. Formally, for a finite alphabet O we denote by O^* the free monoid generated by O by the operation of concatenation, with identity element λ. As usual, $O^+ = O^* \setminus \{\lambda\}$. A multiset S over alphabet O can be represented by a string $x \in O^*$ such that $|x|_a = |S|_a$. For a string or multiset S and $a \in O$, $|S|_a$ denotes the multiplicity of occurrences of a in S.

Definition 2. *A morphogenetic system (*M system*) in* $\mathbb{R}^d$ *is a tuple*

$$\mathcal{M} = (F, P, T, \mu, R, \sigma),$$

where

$F = (O, m, \rho, \epsilon)$ *is a catalogue of floating objects, where*
- O *is a set of floating objects;*
- $m : O \longrightarrow \mathbb{R}^+$ *is the* mean mobility *of each floating object;*
- $\rho : O \longrightarrow \mathbb{R}_0^+$ *specifies the* radius *(of interaction) of the floating objects in* O*;*
- $\epsilon : O \longrightarrow \mathbb{R}_0^+$ *likewise gives the (initial)* concentration *of each floating object in the environment;*

P *is a set of protions;*

$T = (Q, G, \gamma, d_g, S)$ *is a polytopic tile system in* $\mathbb{R}^d$*, with* O*,* P*,* Q*,* G *all pairwise disjoint;*

μ *is the mapping assigning to each tile* $t \in Q$ *a multiset of protions placed on* t *together with their positions:* $\mu(t) \subset P \times \Delta$ *where* Δ *is the underlying polytope of* t;

R *is a finite set of reaction rules;*

$\sigma : \gamma \longrightarrow O^*$ *is the mapping assigning to each glue pair* $(g_1, g_2) \in \gamma$ *a multiset of floating objects which are released to the environment when a connection with glues* (g_1, g_2) *is established.*

A *reaction rule* from the set R has the form $u \to v$, where u and v are strings/multisets which may contain floating objects, protions, glues and/or tiles as specified below. A rule $u \to v$, is applicable when each floating object $o \in u$ is located within the radius $m(o)$ from the reaction site, and eventual further conditions specified by the rule type are also met. Reaction rules can be of four types: metabolic, creative, destructive and dividing.

Metabolic rules
Let $u, v \in O^+$ be non-empty multisets of floating objects and $p \in P$ be a protion placed on a tile.

Type	Rule	Effect
simple	$u \to v$	objects in multiset u react to produce v
catalytic	$pu \to pv$	objects in u react in presence of p to produce v;
	$u[p \to v[p$	this variant requires both u, v at the side "out";
	$[pu \to [pv$	this variant requires both u, v at the side "in";
symport	$u[p \to [pu$	passing of u through protion channel p
	$[pu \to u[p$	to the other side of the tile
antiport	$u[pv \to v[pu$	interchange of u and v through protion channel p

Creation rules $u \to t$,
where $t \in Q$ and $u \in O^+$, can create a tile t while consuming the floating objects in u. Furthermore, t must be able to connect to an existing fixed object at some of its connectors.

Destruction rules $ut \to v$,
where $t \in Q$, $u, v \in O^+$ would destroy a tile t, while consuming the floating objects in u and producing floating objects in v.

Division rules $g \overset{u}{—} h \to g, h$,
where $g—h$ is a pair of glues on connectors of two connected tiles, and $u \in O^+$. As an effect of the application of the rule, the two connectors disconnect and the multiset u is consumed.

Computation of the M system
A *configuration* of an M system is determined by

- the set of all tiles in the environment and their relative positions;
- an interconnection graph of connectors on these tiles;
- positions for all floating objects modulo their mobility.

Configurations with any two objects (tiles or floating objects) in the same or overlapping positions in space are not allowed. The *initial* configuration contains only (unconnected) seed tiles in S and a random distribution of floating objects given by a concentration ϵ_0.

An M system transits from configuration to configuration in discrete time steps by applying rules in its set R. At each step, each floating object can be subject to at most one rule, each connector can be subject to at most one creation or division rule, and each tile can be subject to at most one destruction rule. The rules are chosen and applied in a maximally parallel manner. Finally, each floating object o changes randomly its position according to the Maxwell-Boltzmann distribution with mean mobility $m(o)$.

3 Cytos - simulator of morphogenetic systems

This section briefly describes the M systems simulator called Cytos, its basic modules and main functionality. Architecture of the system is divided into separated modules which cooperate to produce simulation results. The goal of this motion is to make the dynamics of M systems more likely to reproduce macro-properties observed in actual phenomena being modeled, while preserving their computational feasibility. (The Boltzmann distribution is a well know physical models of particle random diffusion in a gas or liquid [4].)

A modular architecture has been chosen for simpler development and maintenanc. Cytos consists of two main modules, namely the simulation and the visualization engine, the visualization module being described in section 3.5 below. The simulation engine is built as a standalone Microsoft Windows DLL (Dynamic Link Library) with a friendly and well described API (application programming interface). The definition of a target M system M is expected by the simulation engine in XML format. The output consists of a (discrete) sequence of states of the system M after a pre-specified number of simulation steps. Each state consists of full information about changes in the previous state in that step. This information is then fed to the visualization engine.

All modules are covered in a single application with a simple user interface called *Cytos*. The package affords full functionality needed for a variety of experiments since it is intended to be a universal simulation tool for M systems, as given by their definition. Figure 1 shows a dependency graph of the mmajor components of Cytos. All components (modules) are written in C# and .NET 4.5.2 (more about these modules below.) The hole project is going to be publish under Open source license, however right now we offer only pre-packed binaries available at `http://sosik.zam.slu.cz/msystem/`. Later this year we are planning to release an M System simulation engine as a standard NuGet package.

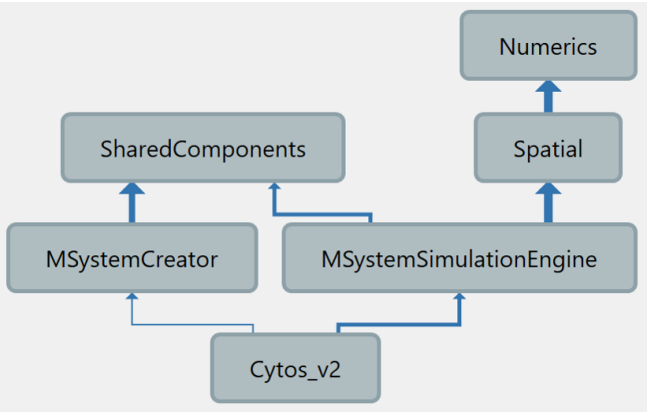

Fig. 1. Dependencies among Cytos's major components.

3.1 Input and output formats

M systems have been formalized in Def. 2. From an implementation standpoint, they need to be translated into XML format. This XML file must give a full specification used as an input for Cytos, called *M system description*. Valid XML format is covered in an XSD scheme used for XML validation. The M system definition has two parts - *tiling* (contains tiles, glues, glue relations and initial objects) and *Msystem* (contains floating objects, proteins, proteins on tiles, evolution rules and signal objects). The output file produced by the simulation engine is also in XML and it is called the *Snapshot file*. It contains discrete computational steps with simulation objects, their positions and states (create, destroy), serving as flags for visualization (an object should be created or erased).

3.2 Cytos

Cytos offers an intuitive and user-friendly user interface (UI.) It is a thin layer over the simulation engine, contains all the functionality required for interaction with the package and is connected with the simulation engine using its own API. This engine is intended to handle simulations of any M system for any project.

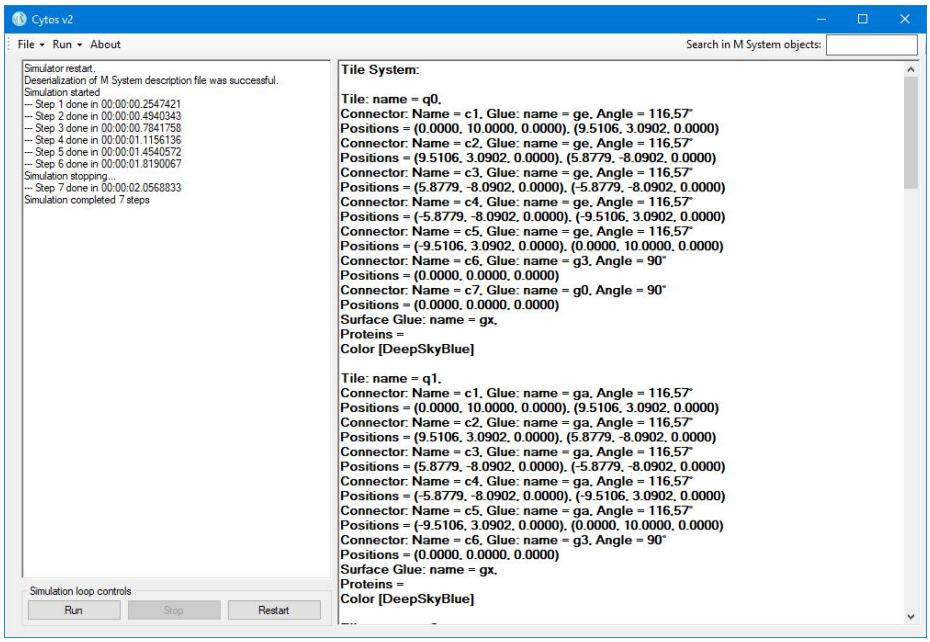

Fig. 2. Cytos main screen.

Figure 2 shows the splash screen of the UI and cosists of two parts. The right pane of the window contains a de-serialized M system description with all

parameters loaded from the input XML. Users can easy search through it using the text box located in the upper right corner; the left pane is an output console to visualize results of different actions, such as de-serialization or simulation runs.

File The main menu offers the user the following options:

- New M system – opens window (module M System creator) for creating a new M system description;
- Load M system – loads a description (in XML) of the M system under simulation;
- Save snapshot – saves computation steps done by simulation to XML, as defined above;
- Visualize snapshot – opens visualization of the current state of the system;
- Open log file – opens basic log file;
- Open simulation log file - opens special simulation log, which contains information about the rules selected for application, number of simulation objects and other information collected in the course of the simulation.

The [Run] menu allows users to start a simulation run and define the number of simulation steps.

3.3 M system creator

An M system description need to be specified in XML format. Although Cytos disposes of XSD scheme (and also XSD validation functionality) for the input XML, it may be cumbersome to find and handle M system properties by direct editing of the XML. For this reason, we have created the module *M system creator*. The Creator contains a clickable GUI for all simulation objects like tiles, glues, floating objects, proteins and also for all types of rules. For each object there is an example, self-validation functionality and useful hints. Users also see the raw XML and they may modify it within Creator application.

3.4 Simulation engine

The simulation engine is the most important and also the most complex module in Cytos. Its functionality is encapsulated into one DLL and it offers all necessary methods for simulation, ranging from input de-serialization to output serialization. The greatest advantage of an independent simulation engine is the possibility to use it in various projects. As a standalone DLL, the engine communicates with other modules through API which provides public methods, properties and events. For a programmer who uses this engine it is not necessary to know how the simulation works, he simply uses data provided by public methods exposed by the API. The main idea is to have a single package that can be repeatedly used “as is” – build easily applications with different purposes powered by the same simulation engine.

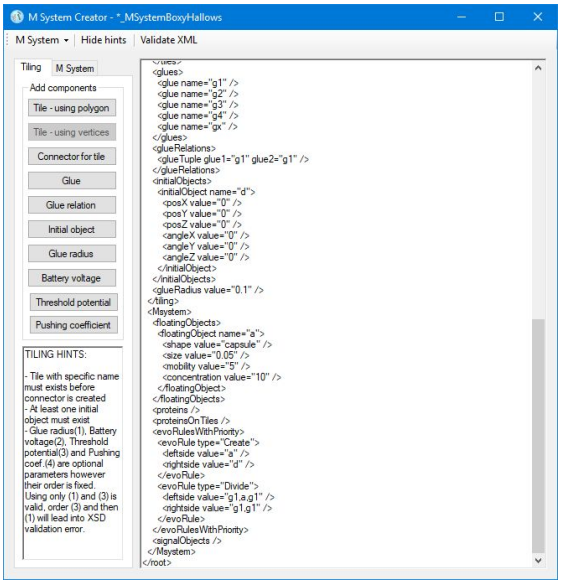

Fig. 3. M system creator - editor to enter the description of a target morphogenetic system.

The M System explicit factors in geometric features. Computing of these features is handled with two mathematical libraries – Math.NET Numerics provides methods and algorithms for numerical computations, and Math.NET Spatial for geometry processing. Math.NET is a pack of open-source numerical libraries written in C# and F#.

3.5 Visualization using unity™

In this project, we made use of the cross-platform game engine Unity developed by Unity Technologies to visualize the output from the simulator. This engine can be used to create a two-dimensional or three-dimensional environment. Furthermore, when an environment is created, it also allows to create a logical layer using the C# programming language. We have chosen Unity for its easy processing, control and availability (free for non-commercial projects).

CytosV is the visualization part of the project that handles the snapshot file from the *Cytos* app. In *CytosV* we can take full advantage of the C# language. There are two ways to run *CytosV*:

1. Run it directly from *Cytos* where we pick from "File Menu" action "Visualize snapshot". At that moment, *Cytos* exports the snapshot file to a temporary folder and runs *CytosV* with parameter path to the temporary snapshot file.

2. Run *CytosV* from the standalone file CytosV.exe located in the same folder as the whole simulator, and use keyboard shortcut Ctrl + O for open file dialog when choosing an exported snapshot file.

Visualization control

The UI uses these main keyboard shortcuts:

- Ctrl + O: Load snapshot file (.xml);
- Ctrl + F1: Show help on screen;
- Ctrl + R: reset environment and turn to first step
- Esc: Close application;
- W: Go straight;
- A: Turn left;
- S: Turn back;
- D: Turn right;
- Left arrow: previous step;
- Right arrow: next step;
- Mouse: Looking around.

Generating simple shapes in 2D and 3D Unity allows to create 2D or 3D objects using so-called assets (3D objects, sounds, textures). There are several ways to generate objects in Unity: either by using empty objects or predefined objects.

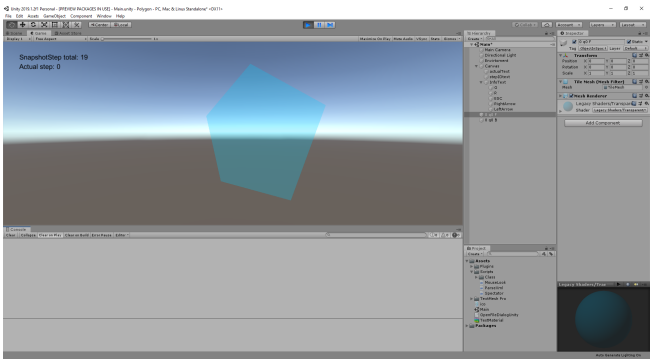

Fig. 4. Unity editor – rendered object pentagon.

When we create objects from an empty object (as in Fig. 4), we need add to this object two important components. The first component is the Mesh Renderer which we use for texture or color on this object. The second one is the Mesh Filter used for modeling in 2D or 3D. A problem with 2D objects in a 3D environment is that the texture is only visible from one side. If we want to

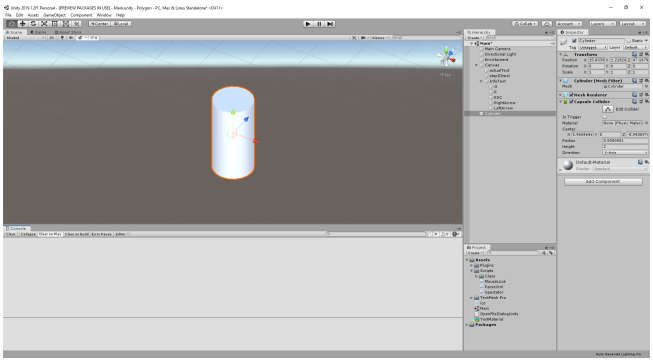

Fig. 5. Unity editor – standard object cylinder.

display the texture from the other side, we must create a copy of this object and rotate it 180°.

In 3D, we can create simple predefined object like spheres, capsules, cubes, cylinders (Fig. 5) or other objects. In Cytos we can use both types of objects. Tiles are rendered on a mesh. In this way, we can render any 2D object like square or pentagon with which we can assemble a 3D object in an environment, such as the dodecahedron shown in Fig. 6. Floating objects in a 3D environment can use standard objects in Unity, like spheres, capsules or cubes.

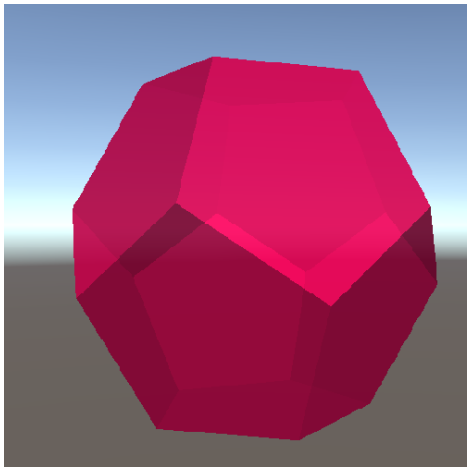

Fig. 6. Unity visualization of a dodecahedron assembled from twelve pentagons.

Preprocessing a snapshot file The process starts by loading the snapshot file into the visualization engine. Once the file is loaded, the total number of steps in the snapshots is calculated. Each snapshot element contains a stepID parameter that specifies an individual step of the simulator. Subsequently, individual steps of simulation are visualized and stored. For example, to create a tile, we need supply the following information:

- Name – specifies the tile name;
- objectID – specifies its unique ID in space;
- type – type of object;
- state – the state of the object (Create, Move, Destroy);
- vertices – determine the points of a given tile in space;
- color – specifies the color of an individual object (RGBA).

An appropriate graphic action is chosen depending on the action to be performed on the object. For example. say "Create", create an nstance of the tile in the environment. An empty object is created with two components (Mesh Renderer and Mesh Filter). Mesh Filter is used for object created from its vertices. Mesh Rendered defines the color texture for the object using attribute called *color* with alpha value. For the action state="Move", the original object is searched for in the environment and then regenerated and displayed in its new position. For the action state="Destroy", the object is searched for in environment and then erased. Hence, we have to render all steps subsequently from the beginning to the desired step. For example, in order to properly render step 4, this step is created by summing actions in all the previous steps (1, 2, 3, 4). This is important for backward stepping in the visualization.

```
<snapshot stepID="0">
<floatingObjects />
<tiles>
  <tile name="q0" objectID="0" type="tile" state="Create">
    <vertices>
    <vertex>
      <posX value="0" />
      <posY value="10" />
      <posZ value="0" />
    </vertex>
    <vertex>
      <posX value="9.51056516295153" />
      <posY value="3.09016994374947" />
      <posZ value="0" />
    </vertex>
    <vertex>
      <posX value="5.87785252292473" />
      <posY value="-8.09016994374947" />
      <posZ value="0" />
    </vertex>
```

```
    <vertex>
      <posX value="-5.87785252292473" />
      <posY value="-8.09016994374948" />
      <posZ value="0" />
    </vertex>
    <vertex>
      <posX value="-9.51056516295154" />
      <posY value="3.09016994374947" />
      <posZ value="0" />
    </vertex>
    </vertices>
    <color name="4000bfff" />
  </tile>
</tiles>
</snapshot>
```

Listing 1. An example of snapshot description of a pentagonal tile in the environment.

4 Examples of simulations in Cytos

In the above paragraphs we have described all important parts and modules of Cytos. This part intends to give the reader answers how to write basic and also move complex simulation examples, how to create different system behaviors and how to visualization results in different steps.

4.1 Boxy hallows

The first and the simplest example is the so called *"Boxy hallows"* (inspired by a scene from the Harry Potter movie). The desire behavior is to create self-reproducing 3D boxes built from six 2D squares. Once the box is completed, it spits into two parts and these parts again complete themselves into boxes. This process repeats until environmental resources are eventually exhausted.

Each side of the box consists of the same tile *d*. The tile is defined by 4 sides and 4 vertices, each side containing the same glue *g1* and the connection angle 90^o. Due to the presence of a single glue, the glue relation used for connecting is obviously *g1-g1*. We also use only one type of floating object *a* with a high concentration which is used for creating tiles *d*.

The dynamics of the model is controlled by just two rules. The creation rule ($a \rightarrow d$) consumes one floating object *a* and it creates tile *d*. The division rule ($g1 \overset{a}{-} g1 \rightarrow g1, g1$) divides glue relation *g1-g1* while consuming one floating object *a*. The initial object is one tile *d*, placed in the environment. Creation rule is applied in the maximally parallel way until the first box is created, then the division rule is used and the process is repeated.

The figure 7 bellow shows the process of creating boxes in simulation steps 1, 2, 4 and 11. Step 1 is the initial configuration, step 2 shows parallel creation

of four sides of box, and the division rule is applied on already created box in step 4. Step 11 is just for illustration how the process continues in a later phase, producing an exponential growth of boxes.

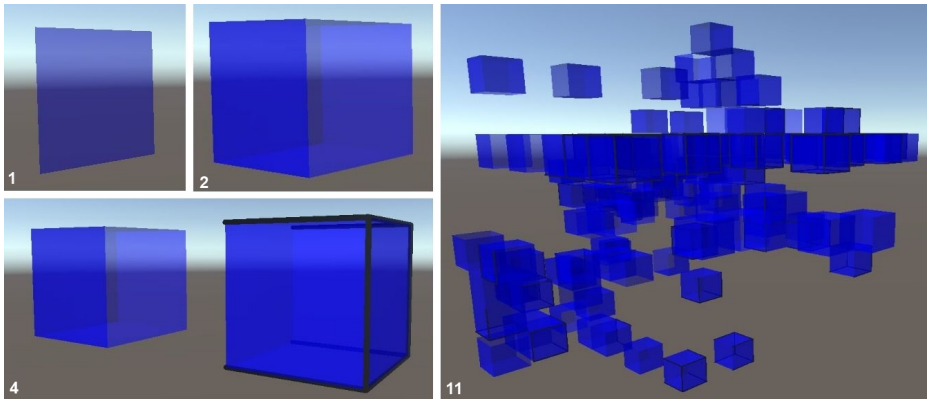

Fig. 7. Visualization of creating boxes in the simulation example "boxy hallows".

4.2 Cytoskeleton-controlled cell division

The second example is a model of cell-like shapes imitating, on a visual level, development of eukaryotic cells with nuclei enclosed in nuclear membranes. Both "cells" and their nuclei are shaped as dodecahedrons consisting of 12 pentagonal tiles. Furthermore, division of these "cells" is controlled by cytoskeleton growth. The division process is triggered by a physical contact of the growing cytoskeleton and nuclear membrane. Although the control of the process and its geometry is nontrivial, the model is described by only 16 local interaction rules. A detailed description of the model and the applied rules is available at `http://sosik.zam.slu.cz/msystem/` or `bmc.memphis.edu/cytos`.

The simulation start with an initial object which is one large pentagon tile. Gradually, the first cell with nucleus is formed and the cytoskeleton starts to grow inside, After a certain phase of growth, the cell (and its nucleus) divides and the process repeats again, consuming environmental resources – floating objects. Figure 8 illustrates the process of cell growth simulation in steps 1, 5, 12, 13, 32. Step 1 contains initial configuration (one large pentagon tile), step 5 shows a complete cell with its nucleus (small red dodecahedron) and interconnecting parts of nucleus. Step 12 shows beginning of the division process of cellular membrane in the environment (black lines). Steps 13 shows a new cell and step 32 demonstrates a part of a growing population of cells. Interestingly enough, the paper [11] demonstrates that the model exhibits properties of robustness and

self-healing, and it is able to develop even under substantial damages incurred to its structure.

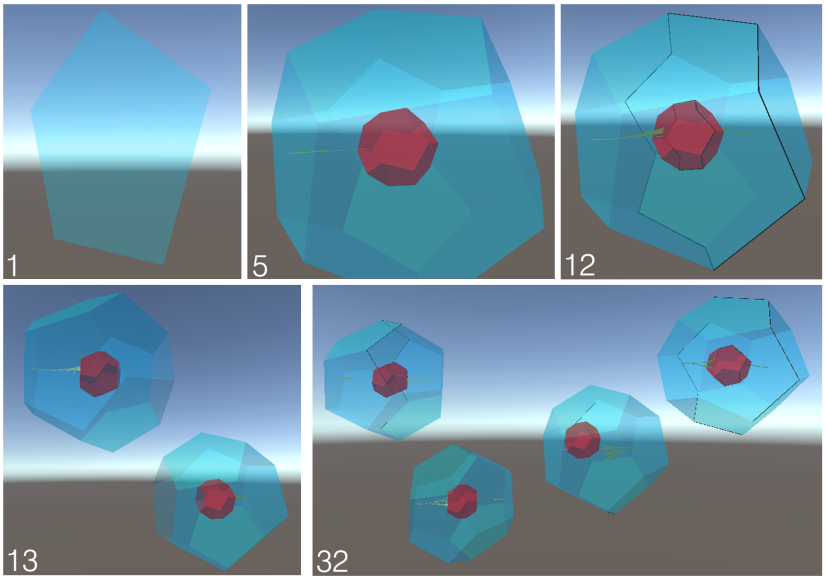

Fig. 8. Growth and division of cell-like forms with "nucleus", where cell division is controlled by a geometrical model of cytoskeleton.

4.3 Cell division with septum

The last example is a simulation of growth of prokaryotic cells without nuclei, performing their fission with so-called *septum*. A septum is a new cell wall, formed gradually between the two nucleoids. Septum extends gradually from the periphery toward the center of the cell. When the new cell walls are complete, the daughter cells separate.

As in the previous example, the cell growth and division consumes resources (floating objects) from the environment, and an exponential growth of cell population continues until these resources are eventually exhausted. The cell fission simulation is simplified in the sense that nucleoids are not included in the simulation, and the fission starts when the cell reaches certain "mature" size. The resulting simulation model contains only 6 creation rules and one division rule. Figure 9 illustrates selected phases of the simulation process, with the number of simulation step indicated in the down left corner. Cells are formed as cuboids with octagonal bases. The first image illustrates details of the septum, remaining images depict various phases of cell fission process.

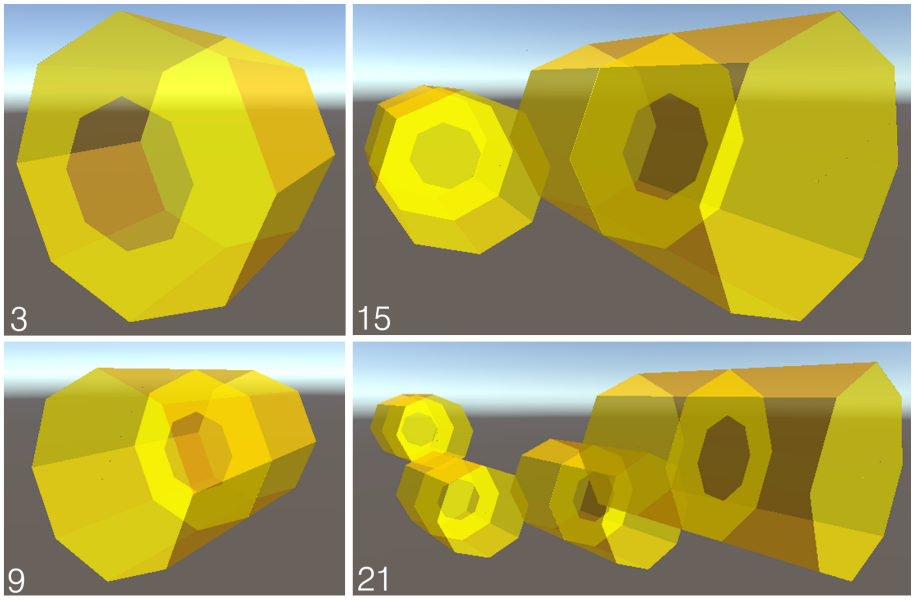

Fig. 9. Septum process

5 Other software simulating membrane systems

As there exist many different varieties of membrane systems, one can find also many applications, programs or simulators. In this section we review a few of them and make a brief comparison with Cytos and its functionality. We include applications still under development or getting get regular updates. Also, we want to point out a universal computation power of membrane systems on diverse application.

With this purpose in mind, we chose applications from four different areas. The first package is P-Lingua, a well known programming language for membrane computing that also includes different tools such as core, compiler and simulator. The second application is MeCoSim. MeCoSim is not a full package but a GUI over a P-Lingua core that simplifies parsing input parameters and generating result files. The third package comes from a completely other area. It uses the power of graphic processing units (GPU) and its huge parallelism to simulate different processes using membrane computing. This GNU (General Public License) project is called PMCGPU and it has been initiated by the Research Group on Natural Computing (Department of Computer Science and Artificial Intelligence, University of Seville). Finally, we mention the simulator Cyto-Sim for stochastic simulations of cellular processes controlled by membranes and (not only) peripheral proteins.

5.1 P-Lingua

P-Lingua is a programming language for P systems with active membranes. As a programming language, it has its own syntax and grammar. A full description of the latest version 4 can be found in [3]). Naturally, P-Lingua is released with a parser, an interpreter and tools supporting its main functionality.

```
/* Initial configuration */
@mu = [[]'2]'1;

/* Initial multisets */
@ms(2) = d{1};

/* Set of rules */
[d{k}]'2 --> +[d{k}]-[d{k}] : 1 <= k <= n;

{
 +[x{i,1} --> r{i,1}]'2;
 -[nx{i,1} --> r{i,1}]'2;
 -[x{i,1} --> #]'2;
 +[nx{i,1} --> #]'2;
} : 1 <= i <= m;

{
 +[x{i,j} --> x{i,j-1}]'2;
 -[x{i,j} --> x{i,j-1}]'2;
 +[nx{i,j} --> nx{i,j-1}]'2;
 -[nx{i,j} --> nx{i,j-1}]'2;
} : 1<=i<=m, 2<=j<=n;
```

Listing 2. Example of P-Lingua, full code can be found here

The main component in P-Lingua is called the pLinguaCore and it is a software framework for cell-like, tissue-like and spiking neural-like P system simulators. PLinguaCore contains a java library and, like Cytos, can be built-in different applications. It is also released with its own translator for input P-Lingua format or XML. There exist several applications which use pLinguaCore and P-Lingua as a main core for simulation, for example MeCoSim [7].

All the examples described above currently use pLinguaCore for simulation. PLinguaCore is a good programmable engine. Its main disadvantage is that all new P system modifications must be firstly implemented there and after that it could be used for a new P system model (or one must implement a new single purpose simulator). In many cases, these new models just use some new regulation or triggering mechanism for an advanced control of multiset operations taking place in its compartments. Therefore, the basic philosophy of P systems (abstract membranes processing multisets of object by simple developmental rules) is usually maintained.

It is a question whether it would be possible to create a platform based on these atomic operations, but still on a higher level than a universal programming language, which would able to cooperate and form hierarchically built applications implementing new P systems models. Another option might be to build an API allowing to include some plug-ins into the pLinguaCore so that the user could introduce new properties without the need to rebuild the whole application.

5.2 MeCoSim

MeCoSim (Membrane Computing Simulator) was introduces as a universal simulation platform for membrane computing. The main goal was to create a fully customizable tool, allowing users to generate custom simulators. As we know from the preceding paragraph, MeCoSim core engine is a pLinguaCore implementing several simulation algorithms for P systems. The latest MeCoSim release includes also the latest pLinguaCore in version 4.0 with support for spiking neural P systems and tissue-like P systems with cell separation rules. However, MeCoSim is still dependent on functionality of pLinguaCore and thus its capability is restricted to the P system modifications implemented in pLinguaCore [7,6].

MeCoSim offers two different user roles: design user and end user. A design user is responsible for defining, debugging and validating the model. An end user creates experiments over the given ecosystem. This partitioning allows users to focus on specific tasks and area of interests [6].

5.3 PMCGPU

PMCGPU is a package that can handle two completely different areas of research. The first one is the membrane computing and bio-inspired computation and the second one is the massive parallel computation on graphic processing units (GPU). Unlike pLinguaCore and MeCoSim, PMCGPU project is written mainly in C/C++ with use of the CUDA architecture.

CUDA (Compute Unified Device Architecture) is a hardware and software architecture developed by nVidia. The main advantage of CUDA is the ability to run instructions in parallel on a GPU. The platform is accessible via the CUDA-accelerated libraries and developers can find it online at the url `https://developer.nvidia.com/gpu-accelerated-libraries`.

One of the key attributes of membrane computing is parallel computation and therefore the use of GPU for computation is a promising approach. As MeCoSim, also PMCGPU uses pLinguaCore for simulation. Although pLinguaCore is a sequential simulator, PMCGPU project handles it in a parallel way. The solution is based on mapping the double parallelism of P systems over the double parallelism of CUDA. Each membrane of the system has assigned a block of CUDA threads and each thread is responsible for one object in the membrane. Using this approach, it is possible to run pLinguaCore in a highly parallel fashion, for details please visit `https://sourceforge.net/projects/pmcgpu/`.

5.4 Cyto-Sim

Cyto-Sim is a stochastic simulator of membrane-enclosed hierarchies of biochemical processes. Its underlying formalism is based on the theory of membrane systems extended to newly defined membrane systems with peripheral and integral protein. The simulator was introduced in [1] and, to demonstrate its potential, applications to model a circadian clock and the G-protein cycle in yeast *saccharomyces cerevisiae* have been presented. Further details of the simulator function and implementation have been described in [2] and [10].

The description of a modelled system and the internal logic of the simulator is based on formal language theory and has been shown to have decidable properties (Cavaliere and Sedwards, 2006), allowing formal analysis in addition to simulation. The simulator provides variable levels of abstraction via arbitrary chemical kinetics which link to ordinary differential equations. The simulation itself is stochastic, using Monte Carlo methods.

The simulator is written in J#, which is part of the .NET framework, and it allows, therefore, porting to both Java and C#. A specific syntax for description of a simulated biological system has been defined. In addition, Cyto-Sim supports models described as Petri nets, can import all versions of SBML and can export SBML and MATLAB m-files. One can find similarities with the Cytos simulator as, for instance, both simulators use an underlying formalism based on the formal language theory and both use stochastic simulation approach. There are also significant differences in their purpose as, e.g., Cyto-Sim does not support directly geometrical features of the modelled system.

6 Conclusions and Future work

In this the paper we have introduced a software package *Cytos* as a simulation tool for morphogenetic (M) systems with graphical output. En route, We have briefly reviewed the model of morphogenetic systems intended to capture computational and algorithmic aspects of morphogenetic phenomena, as observed in biological organisms. The M model partially relies on P systems with proteins on membranes, but fundamentally extends it with self-assembly capabilities, where geometry (shape and position) features play an important role.

We have described the Cytos architecture, its two main modules, the simulation engine and the user interface and its key functionality to provide a platform to explore and understand arbirary M systems. Data from the simulation engine are presented to users using a cross-platform game engine Unity that visualizes all objects in the 2D or 3D environment of the M system. The visualization uses so-called snapshot XML files to describe basic parameters and visual description of one simulation step.

Finally, we have provided a brief comparisons with some existing tools for membrane computing. For instance, Cytos and pLinguaCore offer many similar functionalities. One can note that, as a formal programming language, P-Lingua requires some training for a novice user, on one hand. On the other hand, Cytos

is more like a tool for describing biological processes with simple properties, with the M system creator providing intuitive user interface where the whole description of an M system can be created without further knowledge. It should be noted, however, that the M systems formalism is more complex than that of many P system models, due to the geometrical features included in the model.

As the next steps in the development of Cytos, we would like to improve the visualization in order to show smooth movement of individual objects, including smooth growth of new objects. Further, we would like to rework the overall pre-processing of each step, where we want to optimize loading of the snapshot file and faster start of the simulations. We would also like to transfer this entire project to the web environment to reach more variability and comfort for remote users. Another goal is to incorporate the Cytos simulation engine into the Unity-based visualizer so that we could be able to visualize every step "on the fly" (in real-time) during the simulation.

Acknowledgements

This work was supported by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project IT4Innovations Excellence in Science – LQ1602, and by the Silesian University in Opava under the Student Funding Scheme, project SGS/11/2019.

References

1. Cavaliere, M., Sedwards, S.: Modelling cellular processes using membrane systems with peripheral and integral proteins. In: Priami, C. (ed.) Computational Methods in Systems Biology. pp. 108–126. LNBI 4210, Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
2. Cavaliere, M., Sedwards, S.: Membrane systems with peripheral proteins: transport and evolution. Electronic Notes in Theoretical Computer Science 171(2), 37–53 (2007)
3. Díaz Pernil, D., Pérez Hurtado, I., Pérez Jiménez, M., Riscos Núñez, A.: P-lingua: A programming language for membrane computing. Proceedings of the Sixth Brainstorming Week on Membrane Computing pp. 135–155 (2008)
4. Einstein, A.: Über die von der molekularkinetischen theorie der wärme geforderte bewegung von in ruhenden flüssigkeiten suspendierten teilchen. Annalen der Physik 322(8), 549–560 (1905)
5. Krasnogor, N., Gustafson, S., Pelta, D., Verdegay, J.: Systems Self-Assembly: Multidisciplinary Snapshots. Studies in Multidisciplinarity, Elsevier Science (2011)
6. Pérez-Hurtado, I., Valencia-Cabrera, L., Pérez Jiménez, M., Colomer, M., Riscos-Núñez, A.: MeCoSim: a general purpose software tool for simulating biological phenomena by means of P systems. In: Proceedings 2010 IEEE 5th International Conference on Bio-Inspired Computing: Theories and Applications, BIC-TA. pp. 637–643 (2010)
7. P-Lingua Wiki Page. http://www.p-lingua.org/wiki/index.php/Main_Page, accessed: 2019-05-17

8. Păun, A., Popa, B.: P systems with proteins on membranes. Fundamenta Informaticae 72(4), 467–483 (2006)
9. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Oxford (2010)
10. Sedwards, S., Mazza, T.: Cyto-Sim: a formal language model and stochastic simulator of membrane-enclosed biochemical processes. Bioinformatics 23(20), 2800–2802 (2007)
11. Sosík, P., Smolka, V., Drastík, J., Bradík, J., Garzon, M.: On the robust power of morphogenetic systems for time bounded computation. In: Gheorghe, M. (ed.) Membrane Computing, CMC 2017. Lecture Notes in Computer Science, vol. 10725, pp. 270–292. Springer, Berlin (2018)
12. Sosík, P., Smolka, V., Drastík, J., Moore, T., Garzon, M.: Morphogenetic and homeostatic self-assembled systems. In: Patitz, M.J., Stannett, M. (eds.) Unconventional Computation and Natural Computation: 16th Int. Conf., UCNC 2017. Lecture Notes in Computer Science, vol. 10240, pp. 144–159. Springer, Berlin (2017)
13. Tomita, M.: Whole-cell simulation: a grand challenge of the 21st century. Trends Biotechnol. 19(6), 205–210 (2001)
14. Turing, A.: The chemical basis of morphogenesis. Philos. Trans. R. Soc. Lond. B 237, 7–72 (1950)
15. Winfree, E.: Self-healing tile sets. In: J. Chen, N. Jonoska, G.R. (ed.) Nanotechnology: Science and Computation, pp. 55–66. Natural Computing Series, Springer-Verlag (2006)
16. Ziegler, G.: Lectures on Polytopes. Graduate Texts in Mathematics, Springer-Verlag, New York (1995)