Interaction-Based Modeling of Morphogenesis in MGS

Antoine Spicher, Olivier Michel and Jean-Louis Giavitto

Abstract In this chapter, we advocate a *domain specific language* (DSL) approach to overcome the difficulties of modeling and simulating morphogenetic processes. A careful discussion of the design goals of a DSL leads to the development of an experimental programming language called MGS. Its declarative approach is based on the notion of *topological collection* originating from algebraic topology. Topological collections arise naturally when modeling a "dynamical system with a dynamic structure", or $(\mathrm{DS})^2$, as the state of the system. The evolution function of the system is specified by a *transformation*, which is a set of rewriting rules where each rule defines a local interaction. We illustrate these notions through different models of the same morphogenetic process: the growth of a T-shaped form. The objective is to show how a variety of models can be consistently handled within the MGS framework.

1 Introduction

Most research works presented in this book rely on *simulation* to model, explore and analyze the behavior of engineered morphogenetic processes. Computer simulation is a tool of choice, if not the only systematically available one, for their design. In this context, making the implementation of computer simulation faster, easier, and reusable is absolutely crucial. Yet, the modeling and simulation of such systems remain today difficult and error-prone for at least two reasons:

- The spatial organization of morphogenetic systems is dynamic and requires advanced computer representations, often at different scales.

A. Spicher, O. Michel: Algorithmic, Complexity and Logic Laboratory (LACL), CNRS, Department of Computer Science, Université de Créteil, 94010 Créteil, France ({antoine.spicher,olivier.michel}@u-pec.fr). J.-L. Giavitto: Institut de Recherche et Coordination Acoustique/Musique (IRCAM), CNRS, 75004 Paris, France (giavitto@ircam.fr).

- Models of morphogenetic processes require a coupling between *patterning* and *development*: the evolution of a system's spatial structure depends on its state, while at the same time the evolution of its state depends on its spatial structure.

This situation often leads to the development of ad hoc simulation frameworks, which are dedicated to a particular problem and rely on a particular spatial representation. In contrast, this chapter advocates a domain specific language (DSL) approach to cope with the difficulties encountered when elaborating a computer simulation of natural and engineered morphogenesis.

Designing a DSL for the Simulation of Morphogenesis

DSLs are specially tailored programming languages designed to solve problems in a particular domain [40]. To this end, they provide useful abstractions and notations for the domain at hand. They are more attractive than general-purpose languages for programming in a dedicated domain due to their ease of coding, systematic reuse, better productivity, reliability, maintainability, and flexibility [60]. Furthermore, DSLs are usually small and declarative[1] as opposed to imperative.

Compared with *ad hoc* simulation platforms, a language dedicated to the modeling and simulation of morphogenesis offers more expressive power and wider applicability. For instance, it can facilitate the comparison of various morphogenetic models by providing a uniform and consistent environment for their description and simulation. It can also leverage the "know-how" gathered from various models by enabling model sharing, reuse and coupling.

Yet, the DSL approach must also face difficulties similar to the ad hoc approach. While it addresses the need for generality, there is no unifying formalism that can include all morphogenetic processes. The "wholesale" modeling of morphogenesis would mobilize a great number of models of many types (genetic, mechanical, chemical, etc.) requiring a wide range of formalisms (from continuous to discrete, from deterministic to stochastic) and styles (for example, some models adopt a space-centric view, while others rely on an agent-centric view).

Short of integrating this diversity at a theoretical level, it is nevertheless possible to develop a framework to unify *programming* for simulation purposes. The feasibility of code-level unification is based on three key findings:

- Despite the variety of formalisms and styles used in the simulation of morphogenesis, the vast majority of them capture morphogenetic processes as dynamical processes. However, the *structure* of these dynamical systems is itself dynamic.
- Space-centered or entity-centered modeling can be reconciled through an *interaction-oriented* view. The problem is then to offer sufficiently expressive means of specifying sophisticated interactions.

[1] Declarative programming focuses on *what* should be computed instead of *how* it must be done. Objects and constructions are close to the mathematical standards that enable an easier mathematical reasoning on programs. Thus a declarative program is an executable specification not burdened by the implementation details and remains close to the mathematical model.

- The various spatial organizations that underlie the state of a developmental system can be subsumed under the abstract viewpoint of *topological chains*. However, this does not presume the difficulties of achieving an effective implementation.

Based on these observations, the MGS project started in the early 2000's with the goal of elaborating a DSL dedicated to the modeling and simulation of morphogenesis [21]. Since then, the notions investigated in MGS have been validated by the simulation of numerous examples of processes involved in morphogenesis, such as self-assembly [52, 27, 5], models of gene regulatory networks [20], systems biology (mainly at a cellular level) [24, 49, 41, 53], synthetic biology [18], plant growth [47] and other natural developmental cases [50, 29].

Outline of the Chapter

In the rest of this introductory section, we elaborate upon the above three findings, as they constitute the rationale behind MGS. However, this discussion is not a prerequisite for understanding the self-contained presentation of the MGS language exposed in Section 2. This presentation focuses on the notions of *topological collections* and *transformations* used to represent respectively the state of a morphogenetic process and its evolution function. Section 3 illustrates these constructions with different models of the same example: the development of a T-shaped form. The objective is to show how a variety of models can be handled consistently within the MGS framework.

1.1 (DS)^2*: Coupling Patterning and Development*

In his famous last publication from 1952 entitled " The Chemical Basis of Morphogenesis " [59], Alan Turing elaborates a dynamical systems view of morphogenesis, where he characterizes a developing organism as a set of variables that change over time and capture the state of the system along with its developmental changes. He also conducts a study of the set of all possible trajectories of this system. With a great vision of the fundamental challenges posed by morphogenesis, he writes:

> The interdependence of the chemical and mechanical data [describing the state of a growing embryo] adds enormously to the difficulty, and attention will therefore be confined, so far as is possible, to cases where these can be separated.

Accordingly, Turing decides to focus his attention on simplified cases in which mechanical aspects can be ignored and chemical aspects are the most significant.

Since Turing, it has become a widespread idea that dynamical systems offer general principles for formalizing, understanding and designing self-organization processes such as the ones encountered in morphogenesis. However, in the great majority of models proposed to describe developmental processes, the *shape* (mechanical data) and the *content* (chemical data) are clearly decoupled, making these models usually fall into two categories[2]:

- formalisms that focus on pattern formation in an initially homogeneous but static substrate, and
- generative formalisms that specify the creation and the evolution of a dynamic shape, irrespective of the processes that may take place within the shape.

Examples of the first category are given by reaction-diffusion systems, activator-inhibitor models, or random boolean networks. They rely on various model of dynamical systems, cf. table 1. Examples of the second category include Lindenmayer systems (L-systems) [35], membrane computing [45], graph grammars [48], self-assembly systems, and so on (see table 2).

Table 1 Some formalisms used to specify structured dynamical systems according to the continuous or discrete nature of time, state variables of the system's components and the underlying space in which the patterning process take place. The heading "Numerical Solutions" refers to explicit numerical solutions of partial differential equations (PDE) and systems of coupled ordinary differential equations (ODE).

C: continuous, D: discrete.	PDE	Coupled ODE	Numerical Solutions	Cellular Automata
Space	C	D	D	D
Time	C	C	D	D
States	C	C	C	D

Table 2 Some generative formalisms used to specify the evolution of a shape, according to the topology used to connect the components of this shape. In a *multiset*, all elements are considered to be connected to each other. In a *sequence*, elements are ordered linearly; this case includes lists and extends also to tree-like structures (list of lists). *Uniform* structures represents a regular neighborhood: for example, in a rectangular lattice (Von Neumann neighborhood), each element has exactly four neighbors. GBF presented in section 2.1.4 are a powerful tool to describe such structures relying on mathematical groups. All these formalisms describe spatial structures that can be pictured accurately by a graph. Beyond graphs, *nD combinatorial structures* are used to define arbitrary connections between components of various dimensions. The MGS language presented in the next section handles this kind of shapes as well as the previous ones.

Topology	Multiset	Sequence	Uniform	Arbitrary Graph	*n*D combinatorial structures
Formalism	membrane system	L-systems	GBF	map L-systems, graph-grammars	MGS

[2] This distinction is somewhat contrived. For instance, cellular automata have been devised to study self-reproduction of distinct entities, but these entities are represented by specific patterns in a predefined medium.

Dynamical Systems with a Dynamic Structure

Naturally, various extensions of existing formalisms have been proposed to address the coupling of patterning and development, and blur the above distinction. For example, the original L-systems have been later extended with the notion of *parameters* [46]. This enabled models of plant growth and differentiation, such as *Anabaena Catenula*, based on a chemical paradigm of reaction-diffusion by activation-inhibition. On the one hand, a growth model specifies where diffusion and reaction are possible while, on the other hand, concentrations of chemicals control growth rate and shape change.

This extended L-system can be seen as a dynamical system, yet the set of variables characterizing its state (i.e., the concentration of the chemicals in each cell of the organism) itself changes in time due to the development of new cells. We call such systems *dynamical systems with a dynamic structure*, denoted in short $(\text{DS})^2$, to stress their specificity [17].

Today, using the widespread availability of inexpensive computing power, it is possible to build computational simulations of very large, coupled models. It remains, however, that the simulation of processes that modify themselves due to their own activity—a distinctive feature of morphogenesis—is a problem of great complexity. From the point of view of programming, a main challenge is to come up with a *local description* of the shape *and* the evolution of the state. In fact, if the set of variables that describes the system cannot be known in advance, it is impossible to specify a *global* evolution function. It does not mean that there is no such function, but simply that it cannot be defined explicitly. This is especially the case when the individual (local) interactions between the system's entities are well characterized but the corresponding global evolution function cannot be deduced from these interactions. The macroscopic (global) evolution of the system must be computed as the "temporal and spatial integration" of all the various local and dynamic interactions between the system's elements.

1.2 Space-Centric, Agent-Centric and Interaction-Oriented Modeling

The seminal article of Turing introduces two models: a set of coupled ordinary differential equations (ODEs) and a formulation based on partial differential equations (PDEs). In the former, each cell is characterized by the concentration of two morphogens and the corresponding equations describe the exchange of morphogens between two adjacent cells due to diffusion and the reactions within a cell. In the latter, there are no cells: the system is described as a continuous medium where morphogens diffuse and react in each point of the domain. In both cases, the spatial structure (whether a set of cells or a continuous domain) is fixed *a priori*. If this where not the case, Turing would have faced the "interdependence" problem that he himself stressed, namely the coupling between the spatial structure and the processes that take place in this domain.

There are two ways of looking at this coupling: from the point of view of the *spatial structure* or from the point of view of the *processes* within this structure. In discrete models, this is the well-known difference between space-centric and agent-based models. For example, a prey-predator model can be instantiated using cellular automata in which the state of a cell represents the presence of a prey, the presence of a predator or the absence of both [15]. This is the spatial viewpoint. The model can also be instantiated by a population of individuals that represent preys or predators, and are able to interact if they are in the same neighborhood. This is the agent-centric viewpoint.

The same distinction arises in continuum mechanics models between the Eulerian and the Lagrangian formulations, see Fig. 1. The Eulerian formulation focuses on what is occurring at a fixed point in a reference frame as time progresses (as in cellular automata). In the Lagrangian formulation, an observer follows the position (and other properties) of a spatial element of the system's structure as it moves through a reference space and time (as in agent-based models).

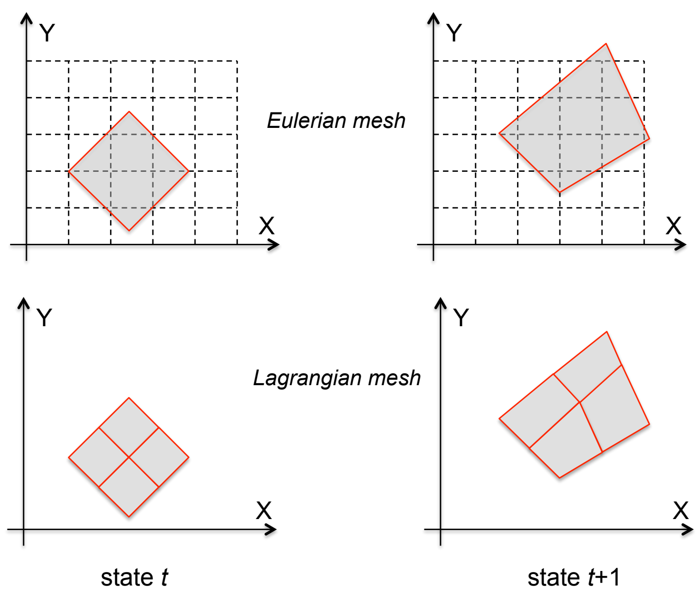

Fig. 1 In fluid dynamics, the Eulerian specification of a flow field describes at each point of a fixed reference frame, the fluid flows as time passes. In a Lagrangian description, the properties (position, stress, etc.) of a piece of a changing shape are followed through time.

Subsuming the Agent-Centric and the Space-Centric Approaches

This distinction has a significant impact on the programming style of the simulation: object-oriented languages and multi-agent systems have been developed to support the agent-centric view, while spatial computing languages [3, 1] support the space-centric view. Leaving apart the difference of entities ("elementary piece of space" vs. "agent"), the essential difference between the two approaches resides in the expression of the system's evolution:

- in the agent-centric view, an entity evolves by receiving a message from another entity, whereas
- in the space-centric view, an entity evolves by querying its neighborhood.

If the spatial structure is static, this neighborhood can be fixed a priori, but if it is dynamic, then the difference between the two points of view starts to vanish. "Elementary pieces of space" become agents that can be dynamically created and rearranged through time. Thus with respect to the simulation of morphogenesis, the two styles diverge only in the expression of the local evolution of an elementary entity: triggered by the entity itself or triggered by another entity.

Neither framework is satisfactory because they both focus on the local evolution of a single entity, which is not expressive enough. For example, to handle the problem of collision of particles in cellular automata, one must either consider a two-phase evolution step (propagation and collision) [54] or turn to lattice gas automata, a variant that considers the coupled evolution of more than one cell [9]. In agent-based modeling, the problem of coordinating synchronously several agents also leads to the development of more flexible schemes [38, 37, 33].

We propose to overcome the limitation of both agent-based and space-based approaches by focusing on the *interactions* between entities (whether agents or elements of space). Interactions specify the simultaneous evolution of a (usually small) subset of the entities composing the system, cf. Fig. 2.

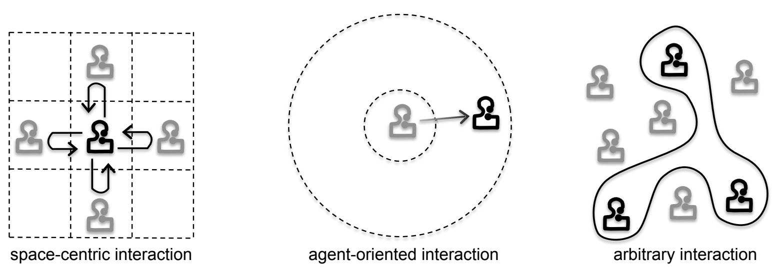

Fig. 2 In space-centric interaction (left), the state of a spatial element (in black) evolves following the state of its neighbors (in gray). In agent-based modeling (middle), the evolution of an agent *A* (in black) is triggered by another agent (in gray) that has *A* in his acquaintances. In these two cases, an elementary evolution is restricted to one entity. More generally, interaction-based modeling enables the simultaneous evolution of a subset of entities (right).

1.3 A Unifying View on Spatial Organization

Viewing a system through the interactions of its elements, instead of its decomposition into elements or the location of these elements, brings forth a new structure [22, 25, 26]. The main idea is the following: if an element s interacts with a subsystem $S_i = \{s_1, \ldots, s_n\}$ (a subset of other elements), then it also interacts, at least conceptually, with any subset S'_i included in S_i. This is a *closure* property, which derives an abstract structure from the subsets of elements. These subsets form a *lattice*, and it is possible (and fruitful) to view this lattice in topological terms as an "abstract simplicial complex" [25], refer to Fig. 3.

The notion of cellular complex is introduced more formally in the next section. It corresponds to a space built by appropriately gluing cells[3], which represent elementary pieces of space of various dimensions. This space is very abstract: it consists of subsets of elements that compose the system and are involved in local interactions. We call this space $\mathscr{A}$ to distinguish it from $\mathbb{R}^3$ where the elements are physically located. Spaces $\mathscr{A}$ and $\mathbb{R}^3$ are a priori different. However, if the interactions satisfy a *locality property*[4], only elements that are neighbors in physical space $\mathbb{R}^3$ can interact directly. In this case, space $\mathscr{A}$ is strongly related to $\mathbb{R}^3$.

We can illustrate via several examples the relevance of considering arbitrary subsets of basic elements as "cells" with a given dimension, both in space-based and in agent-based descriptions. Some physical quantities are naturally linked with pieces of space of a given dimension. For instance, chemical concentration, heat, magnetic

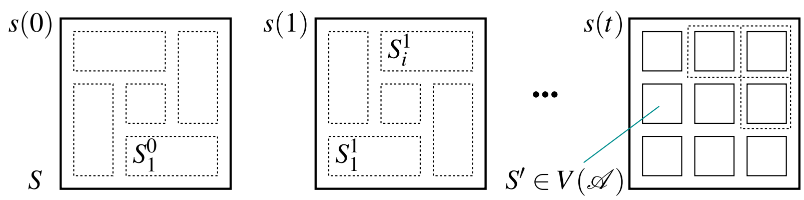

Fig. 3 The interaction structure of a system S resulting from the subsystems of elements in interaction at a given time step. In the interaction view, the decomposition of a system S into subsystems $S_1, S_2, \ldots, S_n$ is *functional*: state $s_i(t+1)$ of the subsystem S_i depends solely of the previous state $s_i(t)$. However, the decomposition of S into the S_i can depend on the time steps. So we write $S^t = \{S^t_1, S^t_2, \ldots, S^t_{n_t}\}$ for the decomposition of the system S at time t and we have: $s_i(t+1) = h^t_i(s_i(t))$ where the h^t_i are the "local evolution functions" of the S^t_i corresponding to an interaction between the elements of S^t_i. The successive decomposition $S^t_1, S^t_2, \ldots, S^t_{n_t}$ can be used to capture the *elementary parts* defined by the smaller sets closed by intersection that contains the S^t_j. These parts corresponds to the agents in agent-based modeling or to elementary pieces of space in space-centric approaches. The *interaction structure* $\mathscr{A}$ is the lattice of inclusions. The leaves $V(\mathscr{A})$ of the lattice are the elementary parts of S.

[3] The term "cell" refers here to a topological notion, not to a biological cell.

[4] The locality property states that matter/energy/information transmissions happen at a finite speed. This property is not always relevant, even for physical systems, for instance because processes may occur on two distinct time scales: changes on the fast time scale may appear instantaneous with respect to the slow time scale, enabling the interaction of arbitrarily far elements.

or electric charges are all associated to volumes; flux are associated to surfaces; tension to lines; temperature, potential, displacement are associated to points. In a biological tissue, the biological cells are volumes that exchange signals through their 2D manifold membranes. The shape of the membrane is constrained by the cytoskeleton which is a 1D branching structure. Signaling between cells is achieved by the exchange of molecules (which are points at this level of abstraction; but if one is interested in the way they interact, the DNA or the proteins are themselves linear sequences folded into 3D shapes), and so on.

A Shift in the Perspective

The interaction-based programming style reverses the perspective adopted in the agent-based and spatial approaches[5]. Instead of proceeding by first specifying the elements in the system, one must define the topological structure $\mathscr{A}$ (i.e., the neighborhood relationships) allowing their interactions. The subsystems (in particular, the components of the system) are then identified with the cells of $\mathscr{A}$.

In this framework, the state of a $(\text{DS})^2$ simply corresponds to the assignation of a (local) state to each component. The topology of $\mathscr{A}$ restricts the possible transition functions of a subsystem S, as the current state of S only depends on the previous state of S and its neighbors. Now, however, S is not restricted to be one agent or one spatial element: it can be any arbitrary *population of agents* or an arbitrary *subspace*. Furthermore, the evolution function not only specifies the evolution of local states but also their coupled evolution with the topology itself.

Such a structure is well-known in algebraic topology: the state of a $(\text{DS})^2$ can be represented by a *topological chain* that associates some values with each cell of the cellular complex [32].

Over the past half-century, there have been notable efforts to develop comprehensive formulations from physics and geometry based on topological chains. The use of chains and cochains to structure the modeling and simulation of physical systems can be traced back to at least the 1960's with Branin [6], who applied these notions to network analysis and circuit design. Later, Tonti and co-authors [55, 56] developed comprehensive discrete formulations of physical laws from first principles [39, 57]. Several studies have subsequently developed this approach in the field of physical modeling and computer-aided design (CAD), notably by Shapiro using the Chain programming language [44] and various follow-up works [8, 14, 12, 13]. Chains have also been used in numerical computation as a tool to structure and generalize the notion of "mesh" [4].

One major goal of these studies is to unravel a proper set of definitions and differential operators that make it possible to operate the machinery of multivariate calculus in a finite discrete space. The motivation is to find an equivalent calculus that operates intrinsically in discrete space, without the reference to the discretization of

[5] Although the interaction-based view could also be qualified as "spatial" because of the topological structure of $\mathscr{A}$, the term "spatial" will be used in this section to qualify only space-centered models referring to the physical space.

an underlying continuous process. This line of research is particularly developed in the field of "geometric modeling", with several recent achievements [11, 30].

However, these works do not focus on the modeling of dynamical structures in the way developed here. Their technical apparatus focuses on uniform computations and metric structures, whereas the MGS language (presented in the next section) relies only on *combinatorial structures*. We believe that the combinatorial approach is less constrained, therefore potentially more amenable to algorithmic computations.

1.4 The MGS Approach to the Simulation of Morphogenesis

The discussion in the previous sections can be summarized by a simple slogan: *the specification of morphogenetic processes must be*

- *local,*
- *interaction-based,*
- *on topological chains.*

The idea is to describe the global dynamics of morphogenesis by summing up local evolutions triggered by local interactions that modify quantities associated to the topological cells of a cellular complex as well as the topological organization of these cells. In this setting, two subsystems S and S' do not interact because they are identified *per se* but because they are neighbors. This property enables the potential interaction of components that do not yet exist in the beginning of the simulation and do not know each other at their time of creation.

This approach has been instantiated in an experimental DSL for the modeling and simulation of morphogenesis, called MGS (which stands for " encore un Modèle Général de Simulation " in French, meaning " yet another general model of simulation "). For technical reasons, the algebraic structure needed on topological chains has been relaxed and the resulting structure is called a *topological collection*.

Transformations are used to define the interactions that make the system evolve. Transformations are functions acting on collections and defined by a specific syntax using rewriting rules. The mechanics of rewriting systems are familiar to anyone who has done arithmetic simplifications: an arithmetic expression can be simplified by repeatedly replacing parts of the expression, called *subexpressions*, with other subexpressions. For example, using the rule

$$\frac{x}{y} \cdot \frac{y}{z} \Rightarrow \frac{x}{z}$$

(where x, y and z are pattern variables representing arbitrary non-null numbers), the expression $\frac{7}{3} \cdot \frac{3}{11} \cdot \frac{11}{5}$ can be rewritten by successive applications of this rule as follows:

$$\frac{7}{3} \cdot \frac{3}{11} \cdot \frac{11}{5} \Longrightarrow \frac{7}{11} \cdot \frac{11}{5} \Longrightarrow \frac{7}{5}.$$

A "transformation" generalizes this process to topological collections with rules based on local interactions:

- the left-hand side of the rule (the "pattern") defines the elements in the system that interact;
- the right-hand side of the rule defines the fate of the elements in interaction.

(see Fig. 4). MGS rules are able to specify the evolution of the quantities associated to the cells of the underlying space as well as the cells and their topological organization.

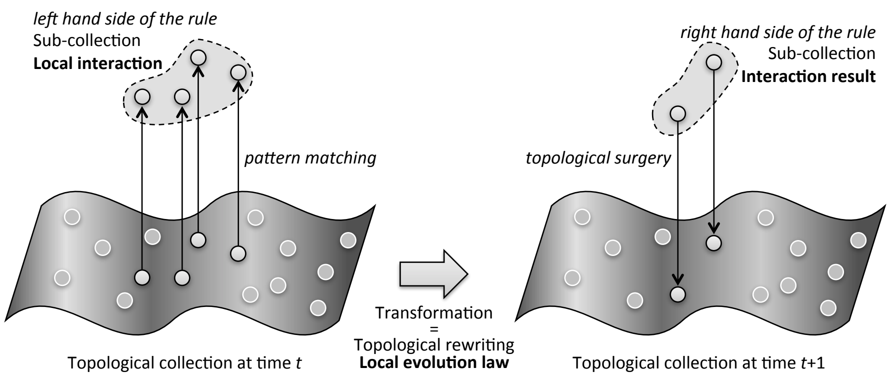

Fig. 4 A topological collection is used to represent the state of a $(\mathrm{DS})^2$ at time t. An MGS transformation gathers one or several rules that specifies the local evolution functions of the $(\mathrm{DS})^2$: the left hand side of a rule defines a sub-collection of elements in interaction using a pattern matching mechanism; the right hand side defines the evolution of this sub-collection. Topological surgery extends the notion of substitution used in rewriting systems to build the new state.

2 A Short Introduction to MGS

MGS embeds the idea of *topological collections* and *transformations* into the framework of a *dynamically typed*, *applicative* language. In our context, dynamically typed means that there is no static type-checking and that type errors are detected at run-time during evaluation. MGS is an applicative programming language: operators acting on values combine values to yield new values, they do not act by side-effect.

Collections constitute the only data structure available in MGS, i.e., the unique way of aggregating values with respect to some neighborhood relationship. Transformations are functions acting on collections and defined by a specific syntax using rules.

2.1 Topological Collections

A topological collection can be viewed as a slight generalization of the notion of *field*. In physics, a field is the assignment of a quantity to each point of a spatial domain [34]. MGS handles spatial domains defined by *abstract cellular complexes* [58].

We start by introducing the notion of abstract cellular complex (Section 2.1.1) and its use to implement topological collections. Then, we present *specific* instantiations of topological collections as different kinds of graphs:

- graphs without neighbors via *records* (Section 2.1.2).
- complete graphs via *(multi)sets*, and linear structures via *sequences* as members of the category of *monoidal collections* (Section 2.1.3)
- regular graphs via group-based fields (*GBFs*, Section 2.1.4),
- irregular graphs via a neighborhood relationship generated by a distance function between elements (*proximal collections* Section 2.1.5),

MGS provides a topological collection based on abstract cellular complexes: this type of topological collections subsumes all the previous one and extend to higher dimensions. However, monoidal collection, GBF, proximal collection, etc., are useful because they correspond to specific topologies and much more efficient implementations.

Finally, we introduce a way to handle multiple collections of the same kind, as it is often the case in nested topological collections, using subtyping (Section 2.1.6).

2.1.1 Abstract Cellular Complex

An abstract cellular complex (ACC) is a formal construction that builds a space in a combinatorial way from simpler objects called *topological cells*. Each topological cell abstractly represents a part of the whole space: points are cells of dimension 0, lines are 1D cells, surfaces are 2D cells, etc. The structure of the whole space, corresponding to a partition into topological cells, is described by *incidence relationships*, which relate a cell to the other cells ***in*** its boundary.

More formally, an abstract cellular complex $K = (\mathsf{C}, \prec, [\cdot])$ is a set C of abstract elements, called *cells*, provided with a partial order $\prec$, called the *incidence relation*, and with a *dimension* function $[\cdot] : \mathsf{C} \to \mathbb{N}$ such that, for each c and c' in C: $c \prec c' \Rightarrow [c] < [c']$. We write $c \in K$ when a cell c is a cell of C. A cell of dimension p is called a p-cell. For example, graphs (which are made of only 0- and 1-cells) are examples of one-dimensional ACCs. Another example is depicted in Fig. 5.

A p-cell c and a q-cell c' are said *incident* if one lies in the boundary of the other, i.e., $c \prec c'$ or $c' \prec c$. Particularly, c is a *face* of c', which is denoted $c < c'$, if they are incident and $p = q - 1$. Conversely, the cell c' is called a *coface* of c. An example of incidence relationship is commented in Fig. 5. More elaborated neighborhoods can be defined from the incidence relationship. In MGS, the (n, p)*-neighborhood* is used: two cells c and c' are *q-neighbor* either if they have a common border of

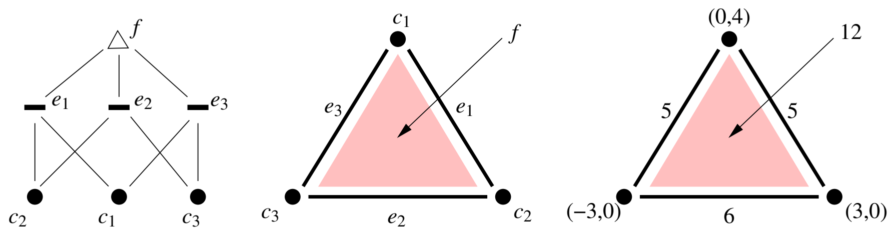

Fig. 5 On the left, the Hasse diagram of boundary relationship of the ACC given in the middle: it is composed of three 0-cells (c_1, c_2, c_3), of three 1-cells (e_1, e_2, e_3) and of a single 2-cells (f). The three edges are the faces of f, and therefore f is a common coface of e_1, e_2 and e_3. On the right, a topological collection associates data with the cells: positions with vertices, lengths with edges and area with f.

dimension q or if they are in the boundary of a q-cell (of higher dimension). If the two cells are of dimension p, we say that they are (p,q)*-neighbors.* A (p,q)*-path* is a sequence of p-cells such that two consecutive cells are *q-neighbors*. For example, the usual notion of "path" in a graph, i.e., a sequence of vertices such that from each of them there is an edge to the next vertex in the sequence, corresponds to the notion of $(0,1)$-path, see Fig. 6.

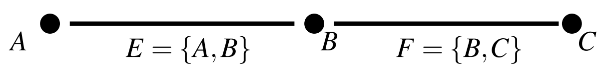

Fig. 6 The 0-cells A and B are 1-neighbor because they are faces of the 1-cell E. The 1-cells E and F are 0-neighbor because the 0-cell B is a common face. The sequence of 0-cells "A,B,C" is a $(0,1)$-path of length 3 and the sequence of 1-cells "E,F" is a $(1,0)$-path of length 3.

Abstract Cellular Complex for Topological Collection

Similar to a field that associates some quantity with the points of a spatial domain, a *topological collection* C is a finite function labeling the cells of an ACC with a value (Fig. 5). Thus the notation $C(c)$ refers to the value of C on cell c. Since a cellular complex may contain an infinite number of cells, we restrict ourselves to collections labeling a finite number of cells. We write $|C|$ for the set of cells for which C is defined. The collection C can be written as a formal sum

$$\sum_{c \in |C|} v_c \cdot c, \text{ where } v_c \stackrel{\text{df}}{=} C(c).$$

With this notation, the underlying ACC is left implicit but can usually be recovered from the context. By convention, when we write a collection C as a sum

$$C = v_1 \cdot c_1 + \cdots + v_p \cdot c_p,$$

we insist that all c_i are distinct. Notice that this addition is associative and commutative. The above notation is used directly in MGS to build new topological collections on arbitrary ACCs of any dimension.

Topological collections are a weakened version of the notion of *topological chain* developed in algebraic topology [42]. They were introduced in [23] to describe arbitrary complex spatial structures that appear in biological systems [24], and other dynamical systems with a time varying structure [17, 28]. From the point of view of computer science, topological collections are reminiscent of *data-fields*, studied e.g. by B. Lisper [31]. Data-fields are a generalization of the "array" data structure, in which the set of indices is extended to finite subsets of $\mathbb{Z}^n$ (see also [19]). With topological collections, in contrast, the underlying space is arbitrary. In fact, the type of a topological collection is determined by the chosen ACC.

Graphs are a particularly important class of ACCs in MGS since it has been showed in [22] how customary data structures (sets, lists, vectors, trees) can be seen as graph-based topological collections: the elements in a data structure are the quantities assigned by the collection to the nodes of a graph; the incidence relationship correspond to the edges of the graph and allows the usual data traversal.

In addition to scalar values (such as symbols, booleans, integers, floats, strings, or lambda-expressions), the current implementation of MGS allows the programmer to handle several types of collections. The elements in a collection can be any type of values, including collections, thus achieving complex objects in the sense of [7]. Through examples, we informally sketch out in the next sections the collection types that we use.

2.1.2 Records

An MGS record is a map that associates a value with a name called *slot*. The value can be of any type, including other records or collections. Accessing the value of a slot in a record is achieved with the dot notation: expression `{ a = 1, b = "red" }.b` evaluates to the string `"red"`. New types of records can be defined using a specific syntax: for instance, `record T = {a:int, b:string }` defines the type `T` of records where slots `a` and `b` are respectively labeled by an integer and a string.

The topologies associated with records are the "totally disconnected" ones: slots in records have no neighbors. Pictured as a graph, records are graph whose vertices are the slots, the labels are the values associated to the slots and there is no edge.

2.1.3 Monoidal Collections

A specific neighborhood relationship that plays an important role in the rest of this chapter is the *full relation*. With this relation, every element in the collection is a neighbor of all the other elements. This corresponds to the *multiset* data structure.

Multisets, with sets and sequences, are called *monoidal collections* because they can be built as a monoid with operator *join*. A sequence corresponds to a join operation that has no special property except associativity, multisets are obtained with an associative and commutative join, and sets when the join operator is associative, commutative and idempotent. The join operator with its properties induces the topology of the collection and the neighborhood relationship: the linear graph for sequences and the complete graph for sets and multisets.

We write `a::m` to add a value `a` in a monoidal collection `m`. The notations `seq:()`, `bag:()` and `set:()` refer to the empty sequence, the empty multiset and the empty set respectively.

2.1.4 The GBF Topological Collection

Topological collections can be defined as *Group Based Fields* (GBF), which are considered as associative arrays whose indices are elements of a group [16]. The latter is defined by a *finite presentation*, i.e., a set of generators together with some constraints on their combination. Thus a GBF can be pictured as a labeled graph where the underlying graph is the Cayley graph of the finite presentation. The labels are the values associated with the vertices and the generators are associated with the edges.

For instance, in order to define a NEWS grid (a rectangular lattice where each node as four neighbors) we may use two generators e (east) and n (north), supporting addition, difference and multiplication by an integer, as illustrated in Fig. 7a.

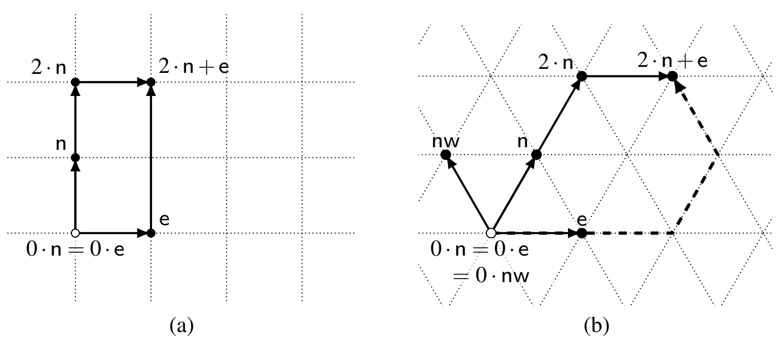

Fig. 7 Left: a GBF defining a NEWS grid, with two generators e and n. Right: a GBF defining an hexagonal grid with three generators e, n and nw, and a constraint $\mathsf{n} - \mathsf{nw} = \mathsf{e}$.

Similarly, an hexagonal grid (6 neighbors for each vertex) can be defined by means of three generators n, e and nw (north-west) and a constraint $\mathsf{n} - \mathsf{nw} = \mathsf{e}$, as illustrated in Fig. 7b. Notice that such grid is adequate to represent cells with a hexagonal shape, since the grid can be paved with hexagons centered on the positions in the grid. As shown by the dashed path, we have $2 \cdot \mathsf{n} + \mathsf{e} = 2 \cdot \mathsf{e} + \mathsf{n} + \mathsf{nw}$, which can be also checked in an algebraic way, by substituting nw with $\mathsf{n} - \mathsf{e}$ in this equality as allowed by the constraint.

The GBF structure is thus adequate to define the arrangement of a grid, in any number of dimensions. A GBF type is specified by the presentation of the underlying group: a list of generators and a list of equations. For example, in the case of the hexagonal grid:

```
gbf H = < n, e, nw; nw+e=n >
```

MGS handles only abelian groups: so the commutation equations are implicit and we use an additive notation.

The relationships between Cayley graphs and group theory are pictured in Fig. 8. A word (a sum of generators) is a path. Path composition corresponds to the group addition. A closed path (a cycle) is a word equal to 0 (the identity of the group). An equation $v = w$ can be rewritten $v - w = e$ and then corresponds to a cycle in the graph. There are two kinds of cycles in the graph: the cycles that are present in all Cayley graphs and correspond to group laws (intuitively: a backtracking path such as $\mathsf{e}+\mathsf{n}-\mathsf{n}-\mathsf{e}$) and closed paths specific to the group's own equations (e.g., $\mathsf{e}-\mathsf{n}-\mathsf{e}+\mathsf{n}$). The graph connectivity (there is always a path going from a node P to a node Q) is equivalent to saying that there is always a solution x to equation $P+x=Q$.

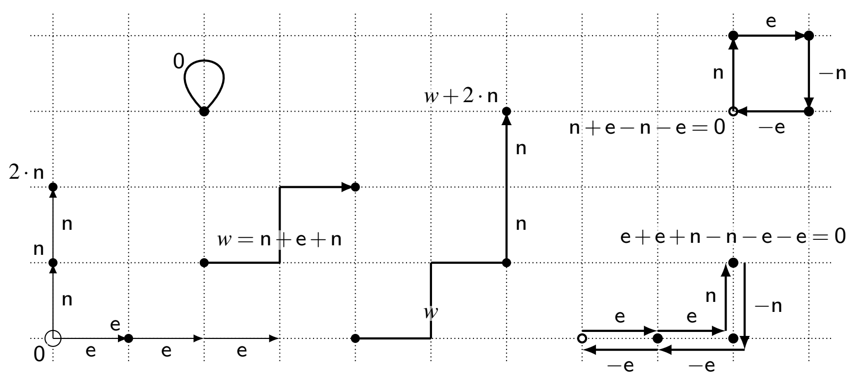

Fig. 8 Graphical representation of the relationships between Cayley graphs and group theory. The GBF pictured is $G = \langle\mathsf{n},\mathsf{e}\rangle$. The equation specifying the commutation between the generators are left implicit. A node in this graph is an element of the group G. A path is a sum of generators and their inverses. The empty path is 0 and correspond to the null displacement. Backtracking paths are closed path in any Cayley graphs. An equation of the presentation corresponds to closed path specific to the group structure. A closed path corresponding to the commutation between n and e, that is $\mathsf{n}+\mathsf{e}=\mathsf{e}+\mathsf{n}$ or in other word $\mathsf{n}+\mathsf{e}-\mathsf{e}-\mathsf{n}=0$ is pictured at the right of the diagram.

2.1.5 Proximal Collections

Proximal collections are graphs whose neighborhood relationship is specified by a relation given on the elements of the collection. For example, let `r` be a relation on integers whose definition is

```
fun r(x, y) = abs(y - x) < 10
```

Then, we are able to define a proximal collection as follows:

```
proximal MyProx = r
```

Two integers `n` and `m` are neighbors in a collection of type `MyProx` if and only if their *distance* (as specified by `r`) is less than 10. We call `r` the *indicator relation* of the proximal collections of type `MyProx`.

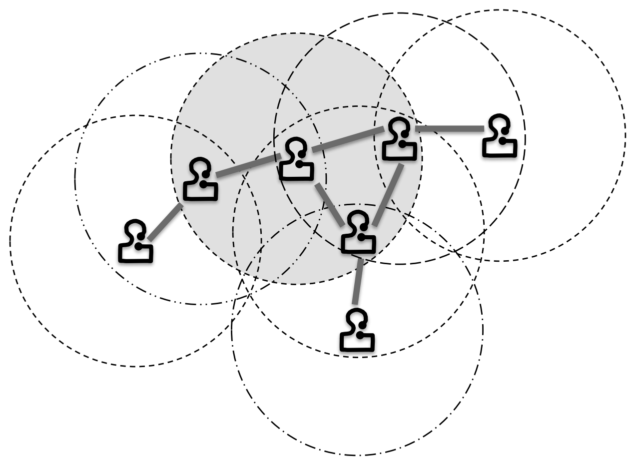

Fig. 9 A *proximal* collection is a set of elements with a relation defining which elements are neighbors. The figure sketches a set of 7 elements located in the 2D plane and two elements are neighbors if the distance between them is below some constant.

2.1.6 User-Defined Subtypes

There is often a need to distinguish between collections of the same type (e.g., several multisets nested in another multiset). This can be accomplished by various means, among which we chose *subtyping*. The subtype of a collection can be thought of as a "color" that does not change the structure of the collection. A collection subtype declaration looks like:

```
collection MySet = set
and collection AnotherSet = set
and collection AnotherMySet = MySet[int]
```

This example specifies a hierarchy of three subtypes: *AnotherMySet* is a subtype of *MySet*, itself a subtype of `set`, and *AnotherSet* is also a subtype of `set` but is not comparable with *MySet*. Remark that the above declaration allows restriction on the types of the elements: a set of subtype *AnotherMySet* can only contain integers.

For each type T, there is an associated predicate with the same name that can be used to check if a value has type T. For example, the expression *MySet*`("this is a string")` returns `false`. Some additional commodity functions are also instantiated when a type is defined: size, pretty-printing, constructors and accessors.

2.2 Transformations

Usually in physics, fields and their evolution are specified using differential operators. MGS generalizes these operators in a rewriting mechanism called *transformation*. A transformation is the application of some local rules following some strategy. A transformation T is written as a set of rules:

```
trans T = { ... rule; ... }.
```

When there is only one rule in the transformation, the enclosing braces can be dropped. A rule is a basic transformation taking the following form:

```
pattern ⟹ expression.
```

It specifies a local evolution of the field: the left-hand side (lhs) of the rule typically matches elements in interaction, and the right-hand side (rhs) computes local updates of the field, see Fig. 4. The application of a local rule $a \Rightarrow b$ in a collection C

1. selects a subcollection A that matches the pattern a,
2. computes a new subcollection B as the result of the evaluation of the expression b instantiated with the collection A, and
3. substitutes B for A in C.

This kind of rewriting removes the part matched by the left hand side and add the part computed by the right hand side.

Transformations are a powerful way to define functions on topological collections that comply with the underlying spatial structure. For instance, a discrete analog of differential operators can be defined using transformations [28]. For multisets, transformations simply reduce to associative-commutative rewriting [10] also called multiset rewriting.

2.2.1 Patterns

We present here a subset of the MGS pattern language. These expressions have a generic meaning, i.e., they can be interpreted in any collection type. The grammar of these pattern expressions is:

$$pat := x \mid \{...\} \mid x\texttt{,}y \mid x < y \mid x\texttt{:}P \mid x/exp$$

where pat, pat' are patterns, x ranges over the pattern variables, P is a predicate and exp is an expression evaluating to a boolean value. The explanation below provides an informal semantics for these patterns.

Variables

A pattern variable x matches exactly one element in the topological collection. The variable x can then occur elsewhere in the rest of the rule to denote the value associated to the matched cell. The notation $\widehat{x}$ is used in the rest of the rule to denote the cell itself. Patterns are *linear*: two distinct pattern variables always refer to two distinct cells.

Record Pattern

The construction $\{...\}$ is used to match a record. The content of the braces can be used to match records with or without a specific field (eventually constrained to a given field type or field value). For instance, $\{a, b : \texttt{string}, c = 3\}$ is a pattern that matches a record with fields a, b of type `string` and c with value 3.

Neighborhood

A pattern is a *composition* of pattern variables. There are three composition operators:

1. The composition denoted by a simple juxtaposition (such as "$x\ y$") does not constrain the arguments of the composition.
2. When two pattern variables are composed using a comma (as in "$x\texttt{,}y$"), it means that the cells matched by x and y must be p-neighbors. The default value for p is 1 and can be explicitly specified during the application of the transformation if needed.
   When the collection is a GBF, it is possible to specify a particular direction for the neighborhood relationship: the pattern x |n> y matches two elements x and y such that if x labels the cell $\widehat{x}$ then y labels the cell $\widehat{x} + \mathsf{n}$.
3. The last composition operator corresponds to the face operator: a pattern "$x < y$" (resp. "$x > y$") matches two cells $\widehat{x}$ and $\widehat{y}$ such that $\widehat{x} < \widehat{y}$ (resp. $\widehat{x} > \widehat{y}$).

Guards

The expression pat/exp matches the subcollections matched by pat verifying exp. Pattern $pat\texttt{:}P$ is a syntactic shortcut for $(pat\ \ \texttt{as}\ \ x)/P(x)$. For instance, x`:int` matches an element x provided that x is an integer and $y/y > 3$ matches an element y provided that $y > 3$ holds (the operator $>$ is overloaded and denote the numeric comparison as well as the incidence relationship).

2.2.2 Rules, Transformations and Application Strategies

A transformation is a set of rules. When a transformation is applied to a collection, the default strategy is to apply the first rule as many times as possible in parallel (a rule can be applied if its pattern matches a subcollection). In the remaining collection, the second rule is applied as many times as possible in parallel with the first rule, and so on. This strategy is the *maximal parallel application strategy* used in L-systems or in Paun systems [45]. Several other strategies are available in MGS such as the *Gillespie application strategy* based on Gillespie's stochastic simulation algorithm used to model chemical reactions [51]. Strategies provide a fine control over the choice of the rules applied within a transformation. They are often non-deterministic, i.e., when applied on a collection C, only one of the possible outcomes (randomly chosen) is returned by the transformation.

A transformation T is a function like any other function and a *first-class* value. For instance, a transformation can be passed as an argument to another function or returned as a result. It allows to compose transformations very easily, leading to a higher-order functional programming style.

The expression $T(c)$ denotes the application of one transformation step to the collection c. A transformation step consists of the application of the rules following the rule application strategy. A transformation step can be effortlessly iterated:

$T[\texttt{iter} = n](c)$	denotes the application of n transformation steps to c
$T[\texttt{fix}](c)$	applies the transformation T until a fixed point is reached

Notice that transformations are first class functions (which are first class value).

3 The growth of a "T" Shape in MGS

In this section, we show MGS at work by developing three different versions of the same problem: the development of a T-shaped structure. This problem has been proposed as a reference example to compare spatial programming languages in [3], especially from the point of view of three paradigmatic tasks: the creation of a coordinate system, the grows of a structure of a given shape and the patterning of this structure. However, we are here more interested in showing the benefits of the MGS approach by showing how the notions of topological collection and of transformations are flexible enough to accommodate various modeling styles and the reuse enabled by the DSL approach. We develop three models:

- the first is based on cellular automata (a regular space-time lattice),
- the second one is based on proximal collection and corresponds to an amorphous medium and an asynchronous evolution,
- our final example develops the T-shape starting from an empty space, using cellular complex in 2D.

In all models, we supose that the T-shapped forms grows starting from one (or few) entities in two successive phases. In the first phase (FGP), the growth process follows a “vertical” direction. In the second phase (SGP), the two horizontal segments of the “T” grow in parallel.

We do not elaborate very much on the acquisition of the “vertical” and the “horizontal” direction: the elaboration of such information of position is not the focus of our example. So, for the cellular automaton we use an anisotropic neighborhood that makes explicit such directions. For the amorphous medium, we rely on the coordinate on each node. For the cellular complex, some walls of the cells are labeled and we rely on this label for the asymetric growth (this labels are explicitly propagated during the growth). Obviously, acquiring an information of position by local means in an initial symmetric medium, is a problem by itself, see for instance [61, 43, 2, 36]. But the implementation of the various algorithms proposed in the literature is straightforward.

All the code presented here are actual MGS programs. The MGS interpreter, as well as numerous other examples are accessible from the MGS web page:
`mgs.spatial-computing.org`

3.1 The “Genetic Device”

This part, shared amongst the 3 models, corresponds to the “control” of the growth. It describes the local state of an entity. The state of an entity gathers two counters $\mathtt{cpt}_V$ and $\mathtt{cpt}_H$. The first is used to manage the duration of the first growth phase (the vertical part of the “ T ”) and $\mathtt{cpt}_H$ the duration of the second growth phase. So a Cell is a record with just two counters.

```
record Cell = { cpt_V, cpt_H }
```

We define a constant called `seed` representing the initial state at time $t = 0$:

```
let seed = { cpt_V = 5, cpt_H = 3 }
```

The subtype `FGP` is a cell with the additional constraint that the counter $\mathtt{cpt}_V$is different from zero :

```
record FGP = Cell + { cpt_V ≠ 0 }
```

MGS types are very expressive : they can be dependant of some values but MGS types are not inferred nor statically type-checked.

A cell is in the second growing phase (`SGP`) if the first counter has reached zero but not the second counter:

```
record SGP = Cell + { cpt_V = 0, cpt_H ≠ 0 }
```

We need also some value to represent an empty location in the spatial organization of cells. We use two ways. In the first, we use simply a symbol `` `Empty``. In the second approach, the empty location is represented by any record wich do not possess the two fields $\mathtt{cpt}_V$ and $\mathtt{cpt}_H$:

```
record Empty = { ~cpt_V, ~cpt_H }
```

To manage the two counters, we rely on the two functions `NextFGP` and `NextSGP`:

```
fun NextFGP (c:FGP) = c + { cpt_V = c.cpt_V - 1 }
fun NextSGP (c:SGP) = c + { cpt_H = c.cpt_H - 1 }
```

The `c:FGP` following the name of the function declares an argument named `c` of type `FGP` (and similarly for `SGP`). The + operator in the body of the functions denotes the asymmetric merge of records. The expression $r_1 + r_2$ computes a new record r having the slots of both r_1 and r_2: $r.a$ has the value of $r_2.a$ if the slot a is present in r_2, else the value of $r_1.a$. The asymmetric merge enables the update of a slot knowing only the slot to update: this makes possible to refine further the record by adding new slots without changing the code already written.

3.2 The Cellular Automaton Implementation

Spatial specification

The cellular automaton based model starts by specifying the underlying lattice. We use in MGS the defining a NEWS grid:

```
gbf NEWS = < north, east >
```

To define an initial population, we use the function `create_NEWS` generated by the previous type definition. The arguments of this function are an initialization function and the range of the generators used to iterate over a set of nodes. More precisely, the expression

```
create_NEWS(f, p, q)
```

creates an instance of the GBF `NEWS` where the node $\mathsf{north}^x + \mathsf{east}^y$ is labeled[6] by the value $f(x,y)$ for $0 \leq x < p$ and $0 \leq y < q$. Hence, the initial state `state0` is computed as:

```
fun init (x, y) = if (x=5 and y=3) then seed else `Empty
let state0 = create_NEWS(11, 11, init)
```

The initial state is a 11×11 grid, where all nodes except one at position $(5,3)$ are labeled by `` `Empty``.

Evolution rules

The evolution of the cellular automaton is specified by two rules. The first correspond to the first growing phase. In this phase, a cell in the `FGP` state extends to a north neighbor if this neighbor is empty:

[6] Note that the labeling of node is not necessarily unique because two different sums may represent the same node (for instance in the hexagonal lattice). However, MGS takes care to visit only once each node in the specified domain even if the “coordinates” of the node are not unique.

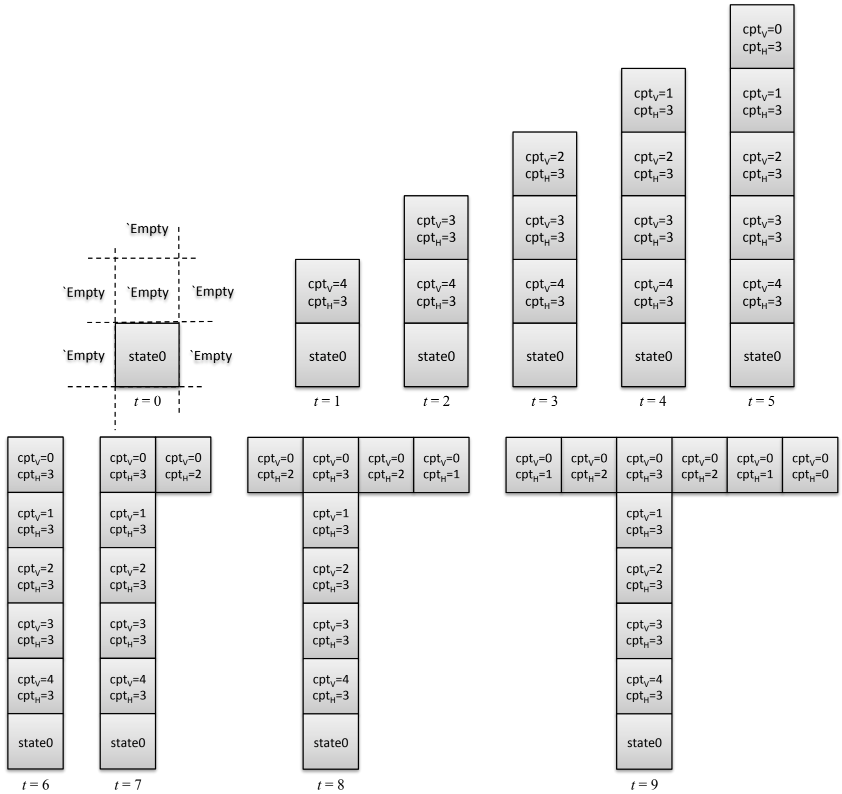

Fig. 10 The first 9 steps of the evolution of the AC. During the transition between state 6 and state 7 the same rule has two exclusive match: dividing a cell to the right or to the left. Because one match exclude the other (the central cell is "consumed" and cannot be used in another rule application at the same step, the horizontal growth is asymmetric (one branch is one step forward the other). Which branch grows first is randomly chosen during the matching. The final step (not figured here) adds a cell to the left branch.

```
trans Rules = {
    c:FGP |north> `Empty ⇒ c, NextFGP(c);
    c:SGP <east> `Empty ⇒ c, NextSGP(c);
}
```

The second rule specifies the evolution during the second growing phase: a cell invades its neighbor to the east or to the west if this neigbor is empty. The syntax <east> refers either to the |east> or to the <east| direction (and <east| $= |-$ east>).

The two elements matched in the left hand side of a rule are replaced by the two elements computed in the right hand side. In this case, the first element `c` remains untouched and the second element changes from `` `Empty `` to a cell with a counter that has decreased w.r.t. `c`.

By default, a transformation applies as much as possible (in space) the rules of a transformation. Thus, one application of a transformation corresponds to one elementary synchronous evolution step of a cellular automaton. The application is iterated to compute the successive states of the systems `Rules[fixpoint](state0)` and a trajectory is illustrated in Fig. 10.

3.3 The Proximal Implementation

Spatial specification

A proximal is a collection of elements whose neighborhood is defined by a predicate. Here, the elements are located in a 2D plane and two points P and Q may interact if their relative distance is below 10.0:

```
proximal AMORPH = fun n -> 100.0 > (n.x * n.x + n.y * n.y)
```

(the right hand side is an anonymous function, lambda expression in MGS notation). By sampling the point randomly, we have the usual amorphous medium [1]:

```
fun init (i, acc) = { x = random(100), y = random(100) }::acc
let state0 = (seed + { x = 50, y = 25 }) :: fold(init, AMORPH:(), 999)
```

The `fold` applies to the integer 999. When an integer p is used where a collection is expected, it corresponds to a cardinal, that is, a set of p elements (from zero to $p-1$). So, the previous fold iterates 999 times. The accumulator is initialized with the value `AMORPH:()` which denotes the empty collection of type `AMORPH`. During the iterations, a record with an random slot `x` and a random slot `y` is added to the accumulator `acc`. The insertion of a new element in the collection is denoted by `::` (such operation exists only for monoidal collections; this is the case of proximals, which are multisets of elements with a dedicated neighborhood function).

Evolution rules

Two auxilliary predicates are used for the evolution function. The predicate `north` is true if its second argument is below the first (following the y direction) and if the distance following x is less than 2. The definition of `east_west` is similar but we restrict only the variation following the y axis:

```
fun north (n1, n2) = (n2.y < n1.y) & 2 > abs(n2.x - n1.x)
fun east_west (n1, n2) = 2 > abs(n2.y - n1.y)
```

The transformation implementing the evolution function has two rules similar to the CA rules:

```
trans Rules = {
    n1:FGP, n2:Empty / north(n1, n2) ⇒ n1, n2 + NextFGP(n1)
    n1:SGP, n2:Empty / east_west(n1, n2) ⇒ n1, n2 + NextSGP(n1)
}
```

Two elements `n1` and `n2` are matched:

- they must be neighbors (the comma operator);
- the first element `n1` satisfies a phase predicate;
- the second element `n2` must be an empty place (that is, a record with no $\texttt{cpt}_V$ nor $\texttt{cpt}_H$ slots);
- `n2` must be at the `north` of `n1` for the first rule and at the `east_west` of `n1` for the second rule. This constraint is achieved by specifying a guard using the " `/` " operator.

In the righ hand side of the rule, the elements that replace the elements matched in the left hand side are computed using an asymmetric merge between records: the counter of `n2` is updated with the counter of `n1` decreased by one. An evolution is illustrated at Fig. 11.

By default, the maximal parallel application strategy is used in the application of the rules. However, it is easy to trigger only one rule application at each transformation application, achieving a fully asynchronous evolution:

```
Rules[fixpoint,strategy=‘asynchronous](state0)
```

The MGS pattern matching device randomize the matching, in order to implement a non-deterministic evolution.

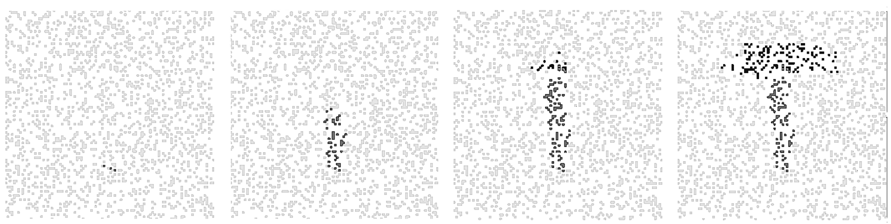

Fig. 11 Initial step (left), two intermediate steps and the final steps (right) of the proximal evolution. Cells in the first growing phase are in dark gray and the cells in the second growing phase are in black.

3.4 The Cellular Complex Implementation

The two previous examples grow the shape in a existing predefined medium: there is no actual true development, rather a patterning process because the building of the shape requires the representation of "empty" places "to invade".

The approach presented here build the shape intrinsically, without relying on the organization of a predefined space. The spatial structure underlying this object is an ACC. and we use 2-cells to represent the basic entities.

Spatial specification

We start from an initial state made of one rectangular face:

```
let state0 =
    letcell v1 = new_vertex()
    and     v2 = new_vertex()
    and     v3 = new_vertex()
    and     v4 = new_vertex()
    and     e12 = new_edge(v1, v2)
    and     e23 = new_edge(v2, v3)
    and     e34 = new_edge(v3, v4)
    and     e41 = new_edge(v4, v1)
    and     f = new_acell(2, (e12,e23,e34,e41))
in
      { x = ..., y = ... } * v1
    + { x = ..., y = ... } * v2
    + { x = ..., y = ... } * v3
    + { x = ..., y = ... } * v4
    + `Basal * e12 + `Apical * e34
    + `Lateral * e23 + `Lateral * e41
    + seed * f
```

The `letcell` construct introduces a recursive definition of related cells. The definition is recursive because if a cell c is the face of c', then c' is a coface of c. The `letcell` takes care to complete the information needed and for example, in the previous statement, we specify only the face of `f` which add implicitly `f` to the coface of the other cells.

The operation `new_vertex` and `new_edge` are short-hand for the general primitive

```
new_acell(dimension [, faces list [, cofaces list]])
```

The lists of faces or cofaces are optional and can be partial.

The previous statement takes the form:

```
letcell ...cell creation and bonding...
in collection-specification
```

The collection specification build a collection using the cells introduced by the `let` (and other cells if needed). The ACC is built using the notation `v*c` which associates the value `v` to the cell `c` and the addition `+` to amalgamate all the labeled cells.

In the previous expression, a record of positions `x` and `y` is associated to the vertices, the edges are labeled by symbols distinguishing 3 kinds of edges, as pictured in Fig. 12 (left). The idea is that the growth takes place on the `` `Apical `` side during the first growth phase and along the `` `Lateral `` side during the second growth phase.

Evolution Rules

We first specify the transformation used to compute the "mechanics" of the system. For the sake of the simplicity, we rely on a very simple mass-spring systems and aristotelician mechanics (that is, the speed is proportional to the force, not the acceleration). This does not change the results because we are interested in the final steady state, but avoid the burden of a double integration.

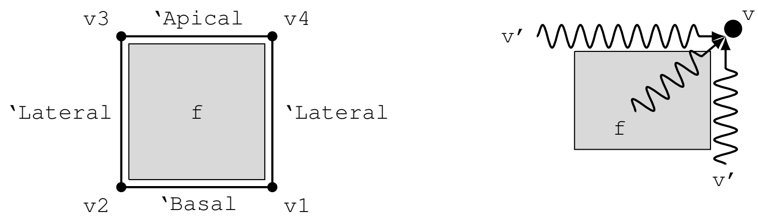

Fig. 12 Left: The initial state is a 2-cell and its faces. Right: a face and the corresponding edges exert a force on the vertices on their boundaries.

Each edge is a spring with a rest length of `L0` and a strength of `k`. The transformation:

```
let sum (acc, v) = { x=acc.x+v.x, y=acc.y+v.y }

trans<2> MecaFace[k,L0,dt] = {
    f ⇒ let n = cardinal(icells(f̂, 0)) in
         let g = icellsfold(sum, {x=0.0,y=0.0}, f̂, 0)
         in f + { x = g.x/n, y = g.y/n }
}
```

matches only 2-cells (this is specified by the `<2>` after the `trans` keyword). The arguments `k`, `L0` and `dt` are additional arguments of the transformation. For each cells `f` the number of 0-cells in its border is computed: `f̂` refer to the cell matched by the pattern variable `f` and the expression `icells(f̂, 0)` returns the set of faces of `f̂` of dimension 0 (“`i`” stands for “incident”). The operator `cardinal` returns the number of elements in the set. The primitive `icellsfold` is used to iterate over the 0-cells in the border of `f̂`. The reduction function is used to compute the barycenter of the corners of the parallelogram `f`.

The next snippet of code integrates the forces and update accordingly the position of the vertexes, cf. Fig. 12 (right):

```
let dist (v, v') = let d1 = v'.x - v.x and d2 = v'.y - v.y
                   in sqrt(d1*d1 + d2*d2)

let sum1 (v, acc, v') =
    let d = dist(v, v') in
    let f = k * (d - L0) / d
    in { x=acc.x+f*(v'.x-v.x), y=acc.y+f*(v'.y-v.y) }

let sum2 (k, v, acc, g) =
    let d = sqrt(dist(g, v) in
    let f = k * (d - sqrt(2.0)*L0) / d
    in { x=acc.x+k*(g.x-v.x), y=acc.y+k*(g.y-v.y) }

trans<0,1> MecaVertex[k,L0,dt] = {
  v ⇒ let Fspring= neighborsfold(sum1(v),{x=0.0,y=0.0},v) in
       let Ftot = icellsfold(sum2(k, v),Fspring, v̂, 2)
       in v + { x=v.x+dt*Ftot.x, y=v.y+dt*Ftot.y }
}
```

The qualifier `<0,1>` means that we focus on 0-cells and that the neighborhood considered in the `neighborsfold` operation is: two 0-cells are neighbors if they are in the border of a common 1-cell i.e., linked by an edge. The variable `Fspring` correspond to the summation of the spring forces. `Ftot` iterates on the cofaces of dimension 2 of $\widehat{v}$ (there can be several such faces, for instance at the junction of the lines of the T). The idea is that each 2-cell exerts an internal pressure from the inside to the outside, constraining the parallelogram to be convex. The initial value of the fold is the forces computed with `Fspring`. The "in" expression, simply integrates the total forces over a time step dt.

The reduction functions `sum1` and `sum2` used in the folds are partially applied: e.g. `sum1` is a function expecting 3 arguments, so `sum1(v)` is a function expecting its last two arguments `acc` and `v'` (provided by the `neighborsfold`).

The full mechanical evolution is given simply by computing one evolution of the face and one evolution of the vertices:

```
fun Meca(ch) = MecaVertex(MecaFace(ch))
```

There is one additional transformation to compute the cell division:

```
trans Extrude = {

   ∼v1 < e12 < ∼f:FGP > e12 > ∼v2
   /(2 = dim(f)) & (e12 = 'Apical)
   ⇒ letcell v3 = new_vertex ()
      and v4 = new_vertex ()
      and e23 = new_edge(v2, v3)
      and e34 = new_edge(v3, v4)
      and e41 = new_edge(v4, v1)
      and e12' = new_edge(v1, v2)
      and f' = new_acell(2, (e12,e23,e34,e41))
      in    (v2 + { x=v2.x + (v2.x-f.x) * random(0.1),
                    y=v2.y + (v2.y-f.y) * random(0.1) }) * v3
         + (v1 + { x=v1.x + (v1.x-f.x) * random(0.1),
                    y=v1.y + (v1.y-f.y) * random(0.1) }) * v4
         + 'Internal * e12'
         + 'Lateral * e23 + 'Lateral * e41
         + 'Apical * e34
         + (NextFGP f) * f'


   ∼v1 < e12 < ∼f:SGP > e12 > ∼v2
   / (2 = dim(f) & (e12 == 'Lateral)
   ⇒ letcell v3 = new_vertex ()
      and v4 = new_vertex ()
      and e23 = new_edge(v2, v3)
      and e34 = new_edge(v3, v4)
      and e41 = new_edge(v4, v1)
      and e12' = new_edge(v1, v2)
      and f' = new_acell 2 (e12,e23,e34,e41))
      in    (v2 + { x= v2.x + (v2.x-f.x) * random(0.1),
                    y=v2.y + (v2.y-f.y) * random(0.1) }) * v3
         + (v1 + { x=v1.x + (v1.x-f.x) * random(0.1),
                    y=v1.y + (v1.y-f.y) * random(0.1) }) * v4
```

```
            + 'Internal * e12
            + 'Basal * e23 + 'Basal * e41
            + 'Lateral * e34
            + (Next SGP f) * f'
}
```

In this code, a pattern ~v matches one element but this element can be matched by another pattern (in the same transformation but not necessarily of the same rule). Contrary to the classical pattern v, such elements are not removed from the result.

The first pattern

```
~v1 < e12 < ~f:FGP > e12 > ~v2 / (2=dim(f)) & (e12='Apical)
```

matches one face f which is a FGP, one edge e12 labeled by the symbol 'Apical and the two vertices bounding e12. The vertices and the face remain in the result but the edge is removed. The right hand side builds several new cells that replace e12 (and updates the neighborhood relationships of the remaining cells).

What is build on the right hand side is a parallelogram (new face f'). The edge e12' replaces the edge e12 and is now lebeled by the symbol 'Internal to prevent further development. The "opposite" edge e23 is labeled by 'Apical in the first growth phase and 'Lateral in the second growth phase.

The labels of v3 and v4 are the labels of v2 and v1 respectively, updated to have new positions. These positions are computed as the previous position displaced by a small random noise: the mechanical evolution will soon unfold the parallelogram in accordance with the internal pressure and the spring force.

The complete evolution of the system is given by composing iteration of the mechanical evolution (here we choose arbitrarily 200 steps) with one growth step.

```
fun Step(ch) = Extrude(Meca[iter=200](ch))
```

Some snapshots of the trajectory are given in Fig. 13.

Acknowledgements The MGS project would not have developed without the participation of, and the fruitful interactions with many colleagues. The authors are pleased to thanks F. Delaplace and H. Klaudel at the U. of Evry, A. Lesne at U. Pierre & Marie Curie Paris, P. Prusinkiewicz at the U. of Calgary, S. Stepney at the U. of York. We especially express our gratitude to the colleagues that make possible the development of the spatial computing initiative (www.spatial-computing.org): J. Beal at BBN, L. Maignan at U. of Paris-Est, F. Gruau at U. of Paris-Sud, S. Dulman at Delft U., R. Doursat at the Institute for Complex Systems in Paris, and many, many others. This research is supported in part by the ANR BLANC SynBioTIC project, the U. of Paris-Est, the UMR STMS 9912 IRCAM - CNRS and the INRIA RepMus team.

References

1. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight Jr, T., Nagpal, R., Rauch, E., Sussman, G., Weiss, R.: Amorphous computing. Communications of the ACM **43**(5), 74–82 (2000)
2. Beal, J., Bachrach, J., Vickery, D., Tobenkin, M.: Fast self-healing gradients. In: Proceedings of the 2008 ACM symposium on Applied computing, pp. 1969–1975. ACM (2008)

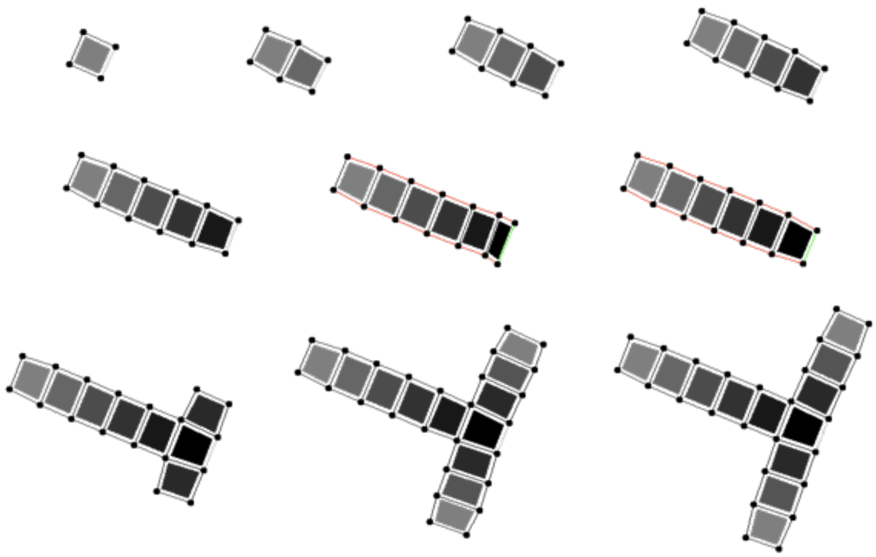

Fig. 13 Initial state (top left), final state (bottom right) and some intermediate states (times goes from left to right and from top to bottom) of the growth of the "T" shape using cellular complex. The last two images in the middle row show the mechanical relaxation between two cell divisions.

3. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. CoRR **abs/1202.5509** (2012)
4. Berti, G.: Generic programming for mesh algorithms: Towards universally usable geometric components. In: H.A. Mang, F.G. Rammerstorfer, J. Eberhardsteiner (eds.) Proceedings of the Fifth World Congress on Computational Mechanics (WCCMV. IACM (2002)
5. Bigo, L., Giavitto, J.L., Spicher, A.: Building topological spaces for musical objects. In: Mathematics and Computation in Music, *LNCS*, vol. 6726. Springer, Paris, France (2011)
6. Branin, F.: The algebraic-topological basis for network analogies and the vector calculus. In: Symposium on generalized networks, pp. 453–491 (1966)
7. Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. Theoretical Computer Science **149**(1), 3–48 (1995)
8. Chard, J.A., Shapiro, V.: A multivector data structure for differential forms and equations. Math. Comput. Simul. **54**(1-3), 33–64 (2000). DOI http://dx.doi.org/10.1016/S0378-4754(00)00198-1
9. Chopard, B., Droz, M., Press, C.U.: Cellular automata modeling of physical systems, vol. 122. Cambridge University Press Cambridge (1998)
10. Dershowitz, N., Hsiang, J., Josephson, N., Plaisted, D.: Associative-commutative rewriting. In: Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 2, pp. 940–944. Morgan Kaufmann Publishers Inc. (1983)
11. Desbrun, M., Kanso, E., Tong, Y.: Discrete differential forms for computational modeling. In: Discrete differential geometry: an applied introduction, pp. 39–54. Schroder, P (2006). SIGGRAPH'06 course notes
12. DiCarlo, A., Milicchio, F., Paoluzzi, A., Shapiro, V.: Solid and physical modeling with chain complexes. In: Proceedings of the 2007 ACM symposium on Solid and physical modeling, pp. 73–84. ACM (2007)
13. DiCarlo, A., Milicchio, F., Paoluzzi, A., Shapiro, V.: Discrete physics using metrized chains. In: 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling, pp. 135–145. ACM (2009)

14. Egli, R., Stewart, N.F.: Chain models in computer simulation. Math. Comput. Simul. **66**(6), 449–468 (2004). DOI http://dx.doi.org/10.1016/j.matcom.2004.02.017
15. Ermentrout, G., Edelstein-Keshet, L.: Cellular autornata approaches to biological modeling. J. theor. Biol **160**, 97–133 (1993)
16. Giavitto, J., Michel, O.: Declarative definition of group indexed data structures and approximation of their domains. In: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming, pp. 150–161. ACM (2001)
17. Giavitto, J.L.: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In: 14th International Conference on Rewriting Technics and Applications (RTA'03), *LNCS*, vol. 2706, pp. 208–233. Springer, Valencia (2003)
18. Giavitto, J.L.: The modeling and the simulation of the fluid machines of synthetic biology. In: M. Gheorghe, G. Paun, G. Rozenberg, A. Salomaa, S. Verlan (eds.) Membrane Computing - 12th International Conference, CMC 2011, Fontainebleau, France, August 23-26, 2011, Revised Selected Papers, *LNCS*, vol. 7184, pp. 19–34. Springer (2012)
19. Giavitto, J.L., De Vito, D., Sansonnet, J.P.: A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$. In: EuroPar'98 Parallel Processing, *Lecture Notes in Computer Science*, vol. 1470, pp. 742–745 (1998)
20. Giavitto, J.L., Malcolm, G., Michel, O.: Rewriting systems and the modelling of biological systems. Comparative and Functional Genomics **5**, 95–99 (2004)
21. Giavitto, J.L., Michel, O.: Mgs: a rule-based programming language for complex objects and collections. In: M. van den Brand, R. Verma (eds.) Electronic Notes in Theoretical Computer Science, vol. 59. Elsevier Science Publishers (2001)
22. Giavitto, J.L., Michel, O.: Data structure as topological spaces. In: Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02, vol. 2509, pp. 137–150. Himeji, Japan (2002). LNCS
23. Giavitto, J.L., Michel, O.: The topological structures of membrane computing. Fundamenta Informaticae **49**, 107–129 (2002)
24. Giavitto, J.L., Michel, O.: Modeling the topological organization of cellular processes. BioSystems **70**(2), 149–163 (2003)
25. Giavitto, J.L., Michel, O., Cohen, J., Spicher, A.: Computation in space and space in computation. In: Unconventional Programming Paradigms (UPP'04), *LNCS*, vol. 3566, pp. 137–152. Spinger, Le Mont Saint-Michel (2005)
26. Giavitto, J.L., Michel, O., Spicher, A.: Interaction based simulation of dynamical system with a dynamical structure. In: P. Kropf (ed.) proc. of the Summer Computer Simulation Conference (SCSC 2011), vol. Track: Modeling and Simulation of Dynamic Structure Systems. The Society for Modeling and Simulation International (SCS) & ACM, Curran assoc. inc. (2011)
27. Giavitto, J.L., Spicher, A.: Systems Self-Assembly: multidisciplinary snapshots, chap. Simulation of self-assembly processes using abstract reduction systems, pp. 199–223. Elsevier (2008). Doi:10.1016/S1571-0831(07)00009-3
28. Giavitto, J.L., Spicher, A.: Topological rewriting and the geometrization of programming. Physica D **237**(9), 1302–1314 (2008). DOI http://dx.doi.org/10.1016/j.physd.2008.03.039
29. Giavitto, J.L., Spicher, A.: Morphogenesis: Origins of Patterns and Shapes, chap. Computer Morphogeneis, pp. 315–340. Springer (2011)
30. Grady, L., Polimeni, J.: Discrete Calculus: Applied Analysis on Graphs for Computational Science. Springer (2010)
31. Hammarlund, P., Lisper, B.: On the relation between functional and data parallel programming languages. In: Proceedings of the conference on Functional programming languages and computer architecture, pp. 210–219. ACM (1993)
32. Hocking, J.G., Young, G.: Topology. Dover publications, New-York (1988)
33. Kubera, Y., Mathieu, P., Picault, S.: Interaction-oriented agent simulations: From theory to implementation. In: Proceeding of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence, pp. 383–387. IOS Press (2008)
34. Leavens, G.T.: Fields in physics are like curried functions or physics for functional programmers. Tech. Rep. TR94-06b, Iowa State University, Department of Computer Science (1994)

35. Lindenmayer, A.: Mathematical models for cellular interaction in development, Parts I and II. Journal of Theoretical Biology **18**, 280–315 (1968)
36. Maignan, L., Gruau, F.: Integer gradient for cellular automata: Principle and examples. In: Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2008., pp. 321–325. IEEE (2008)
37. Mamei, M., Zambonelli, F.: Field-based coordination for pervasive multiagent systems. Springer-Verlag New York Inc (2006)
38. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems. Engineering Societies in the Agents World III pp. 77–98 (2003)
39. Mattiussi, C.: The finite volume, finite element, and finite difference methods as numerical methods for physical field problems. Advances in Imaging and Electron Physics **113**, 1–146 (2000)
40. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. ACM computing surveys (CSUR) **37**(4), 316–344 (2005)
41. Michel, O., Spicher, A., Giavitto, J.L.: Rule-based programming for integrative biological modeling – application to the modeling of the λ phage genetic switch. Natural Computing **8**(4), 865–889 (2009)
42. Munkres, J.: Elements of Algebraic Topology. Addison-Wesley (1984)
43. Nagpal, R., Shrobe, H., Bachrach, J.: Organizing a global coordinate system from local information on an ad hoc sensor network. In: Proceedings of the 2nd international conference on Information processing in sensor networks, *IPSN'03*, vol. LNCS 2634, pp. 333–348. Springer-Verlag (2003)
44. Palmer, R.S., Shapiro, V.: Chain models of physical behavior for engineering analysis and design. Research in Engineering Design **5**, 161–184 (1993). Springer International
45. Păun, G.: Membrane Computing. An Introduction. Springer-Verlag, Berlin (2002)
46. Prusinkiewicz, P., Hanan, J.: Visualization of botanical structures and processes using parametric L-systems. In: D. Thalmann (ed.) Scientific visualization and graphics simulation, pp. 183–201. J. Wiley & Sons, Chichester (1990)
47. Barbier de Reuille, P., Bohn-Courseau, I., Ljung, K., Morin, H., Carraro, N., Godin, C., Traas, J.: Computer simulations reveal properties of the cell-cell signaling network at the shoot apex in Arabidopsis. PNAS **103**(5), 1627–1632 (2006). DOI 10.1073/pnas.0510130103
48. Rozenberg G. an Schürr, A., Winter, A.J., Zündorf, A., Ehrig, H., Kreowski, H.J., Montanari, U. (eds.): Handbook of graph grammars and computing by graph transformation, vol. 1: Foundations, vol. 2: Applications, vol. 3: Concurrency, Parallelism, and Distribution. World Scientific (1997)
49. Spicher, A., Michel, O.: Using rewriting techniques in the simulation of dynamical systems: Application to the modeling of sperm crawling. In: Fifth International Conference on Computational Science (ICCS'05), part I, *LNCS*, vol. 3514, pp. 820–827. Springer, Atlanta, GA, USA (2005)
50. Spicher, A., Michel, O.: Declarative modeling of a neurulation-like process. BioSystems **87**(2-3), 281–288 (2007)
51. Spicher, A., Michel, O., Cieslak, M., Giavitto, J.L., Prusinkiewicz, P.: Stochastic p systems and the simulation of biochemical processes with dynamic compartments. BioSystems **91**(3), 458–472 (2008)
52. Spicher, A., Michel, O., Giavitto, J.L.: Algorithmic self-assembly by accretion and by carving in MGS. In: Proc. of the 7th International Conference on Artificial Evolution (EA'05), *LNCS*, vol. 3871, pp. 189–200. Springer-Verlag (2005)
53. Spicher, A., Michel, O., Giavitto, J.L.: Understanding the Dynamics of Biological Systems: Lessons Learned from Integrative Systems Biology, chap. Interaction-Based Simulations for Integrative Spatial Systems Biology. Springer Verlag (2011)
54. Toffoli, T., Margolus, N.: Cellular automata machines: a new environment for modeling. MIT Press, Cambridge (1987)

55. Tonti, E.: On the mathematical structure of a large class of physicial theories. Rendidiconti della Academia Nazionale dei Lincei **52**(fasc. 1), 48–56 (1972). Scienze fisiche, matematiche et naturali, Serie VIII
56. Tonti, E.: The reason for analogies between physical theories. Appl. Math. Modelling **1**, 37–50 (1976)
57. Tonti, E.: A direct discrete formulation of field laws: The cell method. Computer Modeling in Engineering & Sciences **2**(2), 237–258 (2001)
58. Tucker, A.: An abstract approach to manifolds. The Annals of Mathematics **34**(2), 191–243 (1933)
59. Turing, A.M.: The chemical basis of morphogenesis. Phil. Trans. Roy. Soc. of London **Series B: Biological Sciences**(237), 37–72 (1952)
60. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM Sigplan Notices **35**(6), 26–36 (2000)
61. Wolpert, L.: Positional information and the spatial pattern of cellular differentiation. Journal of theoretical biology **25**(1), 1–47 (1969)