Chapter VI

Modeling Developmental Processes in MGS

Jean-Louis Giavitto
CNRS – University of Évry Val d'Essonne – Genopole, France

Olivier Michel
University of Évry Val d'Essonne – Genopole, France

ABSTRACT

Biology has long inspired unconventional models of computation to computer scientists. This chapter focuses on a model inspired by biological development *both at the molecular and cellular levels. Such biological processes are particularly interesting for computer science because the dynamic organization emerges from many decentralized and local interactions that occur concurrently at several time and space scales. Thus, they provide a source of inspiration to solve various problems related to mobility, distributed systems, open systems, etc. The fundamental mechanisms of biological development are now understood as changes within a complex dynamical system. This chapter advocates that these fundamental mechanisms, although mainly developed in a continuous framework, can be rephrased in a discrete setting relying on the notion of rewriting in a* topological *setting. The discrete formulation is as formal as the continuous one, enables the simulation, and opens a way to the*

systematic study of the behavioral properties of the biological systems. Directly inspired from these developmental processes, the chapter presents an experimental programming language called MGS. MGS is dedicated to the modeling and simulation of dynamical systems with dynamical structures. The chapter illustrates the basic notions of MGS through several algorithmic examples and by sketching various biological models.

INTRODUCTION

The membrane paradigm, DNA computing, molecular computing, and aqueous computing are examples of unconventional models of computation inspired by molecular biology. In this chapter, we focus on a model inspired by biological development at both molecular and cellular levels. We are interested not only in the interactions between the molecules, but also by the assembling and the structural organization that is dynamically created.

Such biological processes are particularly interesting for a computer scientist because the dynamic organization of the involved entities emerges from many decentralized and local interactions that occur concurrently at several time and space scales. The development of biological organisms has for a long time inspired computer science (see, for instance, the notion of cellular automata noted in von Neumann, 1966). More recently, the emerging domains of *amorphous computing* (*www.swiss.ai.mit.edu/projects/amorphous*), *self-healing systems*, or *autonomic computing* (Kephart & Chess, 2003; Parashar & Hariri, 2003) are also directly inspired by developmental processes found at both molecular and cellular levels. Inspired by the description of various developmental processes as changes in a dynamical system, we propose a new computational paradigm that extends the idea of rewriting systems to a broader class of data structures. To investigate this model, we are developing an experimental programming language called MGS.

However, the fertilization of computer science by biological notions (Paton, 1994) is not a one-way process, and biology has imported many concepts developed within computer science such as the notion of programs, memory, information, control, and many others (Stengers, 1988; Keller, 1995). Obviously, new programming paradigms inspired by basic developmental mechanisms will be also an ideal framework to support and help the biologist in analysing and understanding these kinds of biological processes. We illustrate this cross-fertilization by using MGS to simulate various processes of pattern formation.

Outline of the Chapter

In the next section, we will sketch the requirements needed to model developmental processes at both molecular and cellular levels. Our approach to modeling developmental processes is based on the notion of rewriting. This is not really a novelty; although most of the models in this field rely on partial differential equations, string rewriting through the notion of L systems (Lindenmayer, 1968) is now routinely used to model the development of plants, and multiset rewriting constitutes the theoretical foundation of artificial chemistries (*http://ls11-www.cs.uni-dortmund.de/achem*). The section "A Very Short Introduction to Rewriting Systems" presents some fundamental notions of rewriting with an emphasis on the use of rewriting rules to specify the evolution function of a dynamical system.

The usual notion of term rewriting is too restrictive to be easily used to model biological entities. This drawback motivates the development of MGS, an experimental programming language devoted to the simulation of dynamical systems with a dynamical structure. The section "A Quick Presentation of MGS" sketches the basic principles of MGS and outlines how several rewriting mechanisms are unified in the same framework using the notions of *topological collection* and *transformation*. The presentation is restricted to the notions required to understand the examples given in the next two sections.

The next section, entitled "Paradigms of Pattern Formation", focuses on biological system modeling and details some examples of fundamental processes involved in pattern formation:

- Diffusion and beyond (Fick's law and molecular diffusion).
- Boundary growth (the growth of a snowflake and diffusion limited aggregation).
- Growth of a tumor (illustrating the coupling between mechanical forces and cell division).

The examples provided show the ability of MGS to express, in a simple and straightforward way, various patterns of development. They also validate our approach for the modeling and the simulation of dynamical systems with a dynamical structure. A comparison of our approach to related works and some other models of computation concludes the chapter.

BACKGROUND: DEVELOPMENT AND DYNAMICAL SYSTEMS

A Dynamical System with a Dynamical Structure

In the field of developmental biology, one current theoretical framework views the developmental process as changes within a dynamical system (DS). This point of view can be traced back at least to D'Arcy Thompson, Turing, von Bertalanffy, and Waddington, and contrasts with pure genetically programmed and pre-existing plans that await revelation during the developmental process. In the past two decades or so, the concepts and models of nonlinear dynamical systems have been coupled with models of genetic regulations to overcome the genetic/epigenetic debate on the nature of the ontogenetic processes. These models can be seen in the pioneering works of researchers such as Harper et al. (1986), Kaufman (1995), Maynard-Smith (1999), Varela (1979), Wolpert et al. (2002), and Meinhardt (1982).

A developmental process viewed as a dynamical system often presents the distinctive feature of having a dynamic phase space (the phase space is the set of all possible states of the system). Consider a "classical" dynamical system, like a falling stone. This system is adequately described by a position and a velocity. Even if the position and the velocity of the stone are constantly changing in time, the state of this system is always given as a vector in $\mathbf{R}^3 \times \mathbf{R}^3$. We see that the phase space is given *a priori*, and we say that the DS has a static structure.

In contrast, consider the development of an embryo. Initially, the state of the system is described solely by the chemical state c_0 of the egg (no matter how complex this chemical state). After several divisions, the state of the embryo must describe the chemical states c_i of the old and new cells *and also* their spatial arrangement. The number of cells and their organization at a given time cannot be given *a priori*. Moreover, there is a kind of "circular causality" in the evolution of the chemical states and the evolution of the spatial arrangement of the cells: molecules interfere with the placement of the cells (because some molecules make the membranes sticky) and the position of the cells interferes with the diffusion of the molecules (the cells are the medium of the diffusion and create compartments that change the diffusion). Consequently, the exact phase space of the system cannot be fixed before the evolution and development models must state the evolution of the spatial structure jointly with the evolution of the cells states. We say that this kind of DS exhibits a dynamical structure.

In the rest of this chapter we use the abbreviation $(DS)^2$ to mean "dynamical system with a dynamical structure" (Giavitto & Michel, 2003).

The importance of the dynamical structure of a biological system perceived as a dynamical system has often been recognized under several names. *Hypercycle* in autocatalytic networks (Eigen & Schuster, 1979), *autopoeisis* (Varela, 1979), *variable structure systems* in classical control systems (Itkis, 1976; Hung et al., 1993), *developmental grammar* (Mjolsness et al., 1991), and *organization* (Fontana & Buss, 1994) are notions that have been developed to catch and formalize this idea of a changeable structure of a system.

As a matter of fact, the specification of the evolution of a $(DS)^2$ can be very difficult to achieve, and new programming concepts must be developed to help modeling and simulation. Indeed, standard approaches in the formalization of DS do not allow the specification of the phase space or its topology as an observable of the system[1]. This observation has motivated the development of the MGS project.

Towards a Discrete Conceptual Framework

As in many other complex systems in nature, $(DS)^2$ can exhibit coherent behaviors — the parts are coordinated locally and without a centralized control to produce an emergent, organized, stable, and robust pattern. Here, by pattern, we mean the forms or the shapes produced by the development processes in space or in time (Stevens, 1974). For example, the periodic occurrence of a predefined event, such as a chemical concentration repeatedly reaching a given level before decreasing and increasing again, is also an example of a pattern, but in time rather than in space.

What conceptual framework could be used to specify and analyze these developmental design patterns? A quick look at the works in this area shows that although discrete formalizations are present, the mainstream of the contributions are developed in the framework of continuous models. For example, the famous reaction-diffusion model introduced by A. Turing (1952) is described by a set of partial differential equations (PDEs). However, we advocate that discrete models can be used advantageously to model biological development for several significant reasons:

- In the discrete approach, the composition of the design patterns can be studied from an *algebraic* point of view and, more precisely, from *combinatorial* and *generative* points of view to analyze the space of possible patterns. This last approach profits from all of the results and

tools developed in the field of formal language theory and is able to handle the changes of phase space during the evolution.

- The discrete approach is definitely more abstract — it focuses on the fundamental functional and algorithmic properties of pattern formation, without the burden of the "physico-chemical implementation".
- The discrete approach focuses on high-level properties and is well adapted to our knowledge of the biological data. For example, actual DNA chip measurements usually distinguish only between three and eight relevant activation levels for a gene, metabolic fluxes cannot generally be measured *in vivo* for only one cell, etc. In addition, it is more fruitful to have a functional description (e.g., "the cell is in this phase") than a precise quantitative description ("the concentration of this chemical has reached this level"). In fact, the functional description is at last what we are looking for, and the link between the two kinds of descriptions is highly dependent on the experiment, the organism, the environment, etc.
- Development is often described as a succession of discrete morphogenetic events that represent a discontinuity (a cell divides and we have to handle the change of one cell description to two cell descriptions).
- Finally, the continuous framework raises crippling difficulties:

    - Since the equations involved often have nonlinear terms, they can be solved only by numerical methods.
    - The question of the robustness of the solutions of PDEs is completely open, mainly due to the lack of qualitative understanding of such equations.
    - The continuous formalism makes it difficult to express the discrete nature of the biological entities.

From the modeling perspective, the discrete approach is too abstract, leaving unspecified the actual embodiment of the pattern formation. However, despite the increasing amount of biological data, we are far from obtaining the quantitative data needed by realistic continuous models.

So, at this point we are looking for a formal discrete framework able to support the specification of developmental design patterns. At last, the developmental process consists of transformations of the system's parts, if by transformation we understand the appearance and the disappearance of matter and the changes in quality (size, differentiation) or in position of this matter. Then, the global changes of the whole system must be specified as several local

competing transformations occurring in an organized set of simpler entities. This idea of replacing some parts of a structure by other parts specified by local rules recalls the notion of *rewriting*.

A VERY SHORT INTRODUCTION TO REWRITING SYSTEMS

Based on the notion of rewriting systems (Dershowitz & Jouannaud, 1990; Dershowitz, 1993), the emphasis of this section is on the concepts that are relevant in the context of applying rewriting systems to the simulation of dynamical systems (Giavitto 2003; Giavitto et al., 2004).

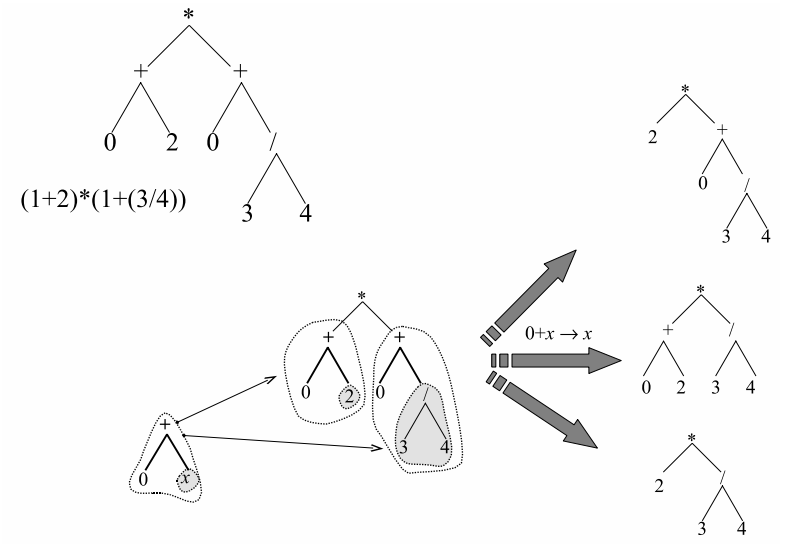
A Computational Device

Computer science has appropriated and developed the notion of rewriting systems (RS) to build and relate different processes. The mechanics of rewriting systems are familiar to anyone who has done high school mathematics: a term can be simplified by repeatedly replacing parts of the term (i.e., subterms) with other equivalent subterms. Terms represent expressions by trees — each internal node is labeled by an operation and the leaves are labeled by constants. A subterm replacement is specified as a rule $\alpha \rightarrow \beta$ where the left-hand-side α specifies a subterm, and the right-hand-side β specifies its replacement. A variable x in α is a kind of wildcard that specifies an entire and unknown subterm; a variable x in β denotes the subterm matched by x in α.

An example involving arithmetic expressions is illustrated in Figure 1. An arithmetic expression is adequately represented by a tree in which the internal nodes are labeled by arithmetic operators and the leaves by integer numbers. An expression with variables is a filter that matches a subtree, the variables catching an entire subtree. A rewrite rule $\alpha \rightarrow \beta$ specifes that a subtree of a given form α must be replaced by a tree β. Several occurrences of a subtree may need to be replaced. In the top-left tree there are two occurrences of a subtree matched by $0+x$. The *rewriting strategy* indicates which occurrence must be replaced. The result of the three possible replacements (replacing one occurrence, the other, or the two) are figured at the right side of the Figure 1.

Let R be a given set of rules, and e a term: we write $e \rightarrow_R e'$ to denote that e can be rewritten in e' using one application of a rule of R. We omit the index R when there is no ambiguity regarding the set of rules used, and we write $\rightarrow^*$ to denote the reflexive-transitive closure of the relation $\rightarrow$. A sequence $e_1 \rightarrow$

Figure 1. Rewriting an arithmetic expression



$e_2 \rightarrow ... \rightarrow e_n$ is called a derivation. The final result of an unextendible sequence of rule applications (i.e., e_n cannot be further rewritten) is called a *normal form*.

RS can be viewed as a kind of directed equations used to compute by repeatedly replacing subterms of a given formula with equal terms until the simplest possible form is obtained. This device has been primarily used as a decision procedure in equational theory: if we can prove that the normal form computed by a RS is unique, then two terms can be proven to be equivalent (modulo the equations corresponding to the rules of the RS) if they share the same normal form. As a formalism, RS has the full power of Turing machines and may be thought of as nondeterministic Markov algorithms over terms, rather than strings.

Rewriting and Simulation

The previous description strongly suggests that a rule $\alpha \rightarrow \beta$ can be used to represent the local evolution of a subsystem α into a new state β. That is, RS can be used for the modeling of a DS, provided that:

- The state of the DS is represented by a term and a subterm represents the state of a subsystem.
- The evolution function of the DS can be specified as the rules of the DS.

For instance, in the context of the development of the embryo, a rule $c \oplus i \to c' \oplus i'$ can be used to specify that a cell in state c that receives a signal i evolves in state c' and sends the signal i'. A rule such as $c \to c' \oplus c''$ defines a cellular division and $c \to \varnothing$ (c gives nothing) represents apoptosis. The idea is that the left-hand-side of a rewriting rule selects an entity in the biological system and the messages that are addressed to it, and the right-hand-side describes the new state of the entity and the eventual messages that are emitted. The operator $\oplus$ that appears in the rule denotes the composition of the entities together with the messages into an entire global system. The ability to equally express the change of state and the appearance or disappearance of entities makes RS suitable to model $(DS)^2$.

The Management of Time

The notion of time that underlies the use of RS for the modeling of DS is clearly based on a discrete atomic event model: time is passing when some event occurs somewhere in the system, a rule application corresponds to the occurrence of such an event, and duration is not handled (but can be emulated using a start and an end event).

A *rewriting strategy* is an algorithm for choosing in a term e the occurrence of the subterm that must be rewritten. Several algorithms can be considered that allow some control over the management of time. For example, intuitively, the laws of nature are encoded into the rules of an RS and these laws must apply everywhere. This leads to consideration of the so-called "parallel application strategy". Instead of considering just one occurrence of a rule application during a rewriting step, one may chose to rewrite in the same step a maximal nonintersecting set S of matching subterms in e: two subterms in S do not have a subterm in common, each element in S matches the left-hand-side of a rule, and S cannot be further extended by a subterm in e without losing the two previous properties. On the other hand, one may suppose that each event occurs at a different time. This asynchronous dynamic fits well with the standard application of only one rule at each rewriting step.

The Management of Space

A rule of form $c \oplus i \to c' \oplus i'$ assumes that signal i produced by some cell reaches its target c, and that signal i' produced by the change of c to c' reaches its target located somewhere in the system. Thus, the operation $\oplus$ used to compose the state of the subsystems and the interaction messages must be able to express the spatial localization and the functional organization of the system.

For instance, suppose that our cells are bacteria in a test tube. The signals produced by one cell are then released in a solution. By thermal agitation, the signals potentially reach any other cell in the test tube. This can easily be achieved by using a (formal) operator $\oplus$ that is *associative* and *commutative*. From this point of view, the term $c_1 \oplus c_2 \oplus c_3$ represents a test tube with three cells. Because we assume that $\oplus$ is associative and commutative, this term can be rewritten in the equivalent term $c_2 \oplus c_1 \oplus c_3$ or $c_3 \oplus c_1 \oplus c_2$. Applying the rule $c_1 \rightarrow c_1 \oplus i$ to this term gives $c_1 \oplus i \oplus c_2 \oplus c_3$. By associativity and commutativity, signal i can be moved to the neighbor of c_3 or any other cell.

Terms built with associative and commutative operators achieve a multiset organization of the objects. A multiset is a set in which an element is allowed to occur multiple times. A multiset builds up a "chemical soup" of elements that are not bound in a tree but can move around and interact with one another.

By imposing only associativity, the term structure reduces to a sequence of elements. So, by giving some properties to the operations in the term, we can represent several kinds of organization. This fact has long been recognized, and multiset rewriting and string rewriting have been successfully used in the field of biological modeling. The rest of this section gives some references of these approaches.

However, multisets, sequences, and trees of elements are far from being sufficient to characterize the sophisticated organization needed to represent the variety of biological structures from molecules to societies, through compartments, cells, tissues, organs, and individuals. This severe shortcoming motivates (among others reasons) the extension of the notions of rewriting to more general structures. In the next section we present such an extension based on topological notions.

Multiset Rewriting

Multiset rewriting is the core of the Gamma language. Gamma (Banâtre & Le Metayer,1986; Banâtre et al., 1987) is based on the chemical reaction metaphor; the data are considered as a multiset M of molecules and the computation is a succession of chemical reactions according to a particular rule. A rule (R, A), where R is a predicate and A a function, indicates which kind of molecules can react together (a subset m of M that satisfies predicates R) and the product of the reaction (the result obtained by applying function A to m). Several reactions may happen in the same time. No assumption is made on the order in which the reactions occur. The only constraint is that if the reaction condition R holds for at least one subset of elements, at least one reaction occurs (computation stops once the reaction condition does not hold for any

subset of the multiset). The CHemical Abstract Machine (CHAM) extends these ideas with a focus on the semantics of nondeterministic processes (Berry & Boudol, 1992).

The application of this abstract chemistry-based approach is now recognized as an emerging field called *artificial chemistries* (Dittrich et al., 2001; *http://ls11-www.cs.uni-dortmund.de/achem*). Various motivations presides the growing body of research done in this field, ranging from the study of the automated generation of combustion reactions (Bournez et al., 2003) to the study of complex dynamical systems and self-organization in biological evolution (Fontana & Buss, 1994).

An important application is the modeling and the study of the behavior of signaling pathways. Fisher et al. (2000) proposed the use of rewriting systems to model cascades of protein interactions in signaling pathways. Later work (Eker et al., 2002; Lincoln, 2003) has produced some very sophisticated models of these pathways; however, the earlier work draws attention to the subtle role that so-called scaffold proteins play in facilitating cascades and preventing cross-talk between pathways. The RS approach provides extremely efficient representations of the information processing nature of signaling pathways. Despite the obvious use of RS for simulating various behaviours, the symbolic tool sets produced (e.g., model checking) can then be applied to generate and check novel hypotheses and to develop an algebra and a logic of signaling pathways.

String Rewriting

If the operator involved in the term to be rewritten is only associative (and not commutative), then the term simply corresponds to a sequence of elements. Such sequences are also called *strings* (especially if the universe of possible elements is finite). String rewriting is a formal framework heavily investigated, for example, in formal language theory.

String rewriting has shown its usefulness and maturity in plant development modeling. Introduced by Lindenmayer (1968), the L system formalism can be roughly described as string rewriting rules applied in parallel to strings representing a linear or a branching structure. The original L system formalism has been extended in many ways, and comprehensive reviews have been produced (Prusinkiewicz & Lindenmayer, 1990; Prusinkiewicz 1998, 1999). It is worth giving a flavor of this formalism through a very simple example.

The two rules, $a \rightarrow ab$ and $b \rightarrow a$ can be viewed as a model of the development of a filamentous organism using the symbols *a* and *b*. The first rule states that a cell of length *a* divides into two adjacent cells of length *a* and *b*. The other rule states that a cell *b* changes its length to *a* (these rules are related to the development of the bacterium *Anabaena* where we have ignored the polarity that would determine if a cell of length *a* divides into *ab* or *ba*). The sequence of the cells is simply denoted by the juxtaposition of their symbols. Starting from a unique cell of length *a*, we obtain successively:

$$a \rightarrow ab \rightarrow aba \rightarrow abaab \rightarrow abaababa \rightarrow abaababaabaab \rightarrow \ldots$$

by applying the two rules in parallel. Each word represents a state of the development of the filament.

A good example of an L system that takes into account the cellular interaction in the development is the modeling of growth and heterocyst differentiation in *Anabaena* (Wilcox et al., 1973; Hammel & Prusinkiewicz, 1996). This example is remarkable for at least two reasons: it shows the ability of this kind of discrete model to accommodate or to easily emulate features usually handled in continuous formalisms (e.g., the modeling of the diffusion) and also because it tackles a fundamental biological mechanism — a morphogenesis driven by a reaction-diffusion process taking place in a growing medium.

By combining and structuring multiset and string rewriting, we can extend the applicability of these formalisms. Applications of such extensions at the genetic level include DNA computing (Adleman, 1994) and splicing systems, a language-theoretic model of DNA recombination that allows the study of the generative power of general recombination and of sets of enzymatic activities (Head, 1987, 1992).

A QUICK PRESENTATION OF MGS

We present the fundamental notions that underlie the MGS programming language. MGS stands for "encore un Modèle Général de Simulation" (or, "yet another general model for simulation"). The notion of *topological collection* developed in MGS enables the unification of various forms of rewriting and its extension to more general data structures than trees.

Topological Collection and Their Transformations

MGS is aimed at the representation and manipulation of local transformations of entities structured by *abstract topologies* (Giavitto & Michel, 2001,

2002). A set of entities organized by an abstract topology is called a *topological collection*. "Topological" here means that each collection type defines a neighborhood relation specifying both the notion of *locality* and the notion of *subcollection*. A subcollection B of a collection A is a subset of elements of A defined by some *path* and inheriting its organization from A. A path is a sequence-adjacent element, the adjacency relation being specified by the neighborhood relationship between the elements of the data structure.

Abstractly, a topological collection can be formalized as a partial function C that associates a value to the points of a discrete topological space. The points of this space are called the *positions* of the collection. The values associated with these positions, or the image of the partial function C, are the elements of the collection. The topological structure associated with the set of positions gives the neighborhood relationship between the collection elements. For example, a 2-D array filled with integer elements can be seen as a partial function from $\mathbf{Z}^2$ to $\mathbf{N}$ with a finite definition domain. Here, the topology of $\mathbf{Z}^2$ is inherited from the module structure of $\mathbf{Z}^2$ (Munkres, 1993) and a position (x, y) is the neighbor of a position (x',y') only if $x'=x\pm1$ and $y'=y\pm1$. This representation of an array is only an abstract view — an array is not really implemented as a function within the computer but as a set of values indexed by positions taken in $\mathbf{Z}^2$.

The *global* transformation of a topological collection C consists of the parallel application of a set of *local* transformations. A local transformation is specified by a rewriting rule r that specifies the change of a subcollection. The application of a rewrite rule $r = \alpha \rightarrow f(\alpha)$ to a collection A:

- Selects a subcollection B of A whose elements match the path pattern α.
- Computes a new collection C as a function f of B and its neighbors.
- Specifies the insertion of C in place of B into A.

In other words, MGS extends the idea of the term by the idea of topological collection and generalizes the notion of the rewriting rule to the notion of transformation. For the sake of the expressivity, MGS embeds the idea of topological collections and their transformations into the framework of a simple dynamically typed functional language. Collections are just new kinds of values, and transformations are functions acting on collections and defined by a specific syntax using rules. Functions and transformations are first-class values and can be passed as arguments or returned as the result of an application. MGS is an applicative programming language: operators acting on

values combine values to give new values, they do not act by side-effect. In our context, *dynamically typed* means that there is no static type checking and that type errors are detected at run-time during evaluation. Although dynamically typed, the set of values has a rich type structure used in the definition of pattern-matching, rules, and transformations.

Collection Types

There are several predefined collection types in MGS, and also several means of constructing new collection types. The collection types can range in MGS from totally unstructured with sets and multisets to more structured with sequences and GBFs (GBFs generalize the notion of regular array and are presented in the section entitled "Group-Based Data Field"). Other topologies are currently under development and include general graphs and abstract simplicial complexes. Abstract simplicial complexes generalize the notion of graphs and enable the representation of arbitrary topology (Munkres, 1993). However, in this chapter we are mainly concerned with three families of collection types: *monoidal collections*, *GBFs*, and *graphs*.

For any collection type T, the corresponding empty collection is written ():T. The name of a type is also a predicate used to test if a value *v* has this type: T(*v*) holds only if *v* is of type T. Each collection type can be subtyped:

```
collection U = T
```

introduces a new collection type U, which is a subtype of T. These two types share the same topology but a value of type U can be distinguished from a value of type T by the predicate U. Elements in a collection T can be of any type, including collections, and thus achieving complex objects in the sense of Buneman et al. (1995).

Monoidal Collections

Sets, multisets, and sequences are members of the monoidal collection family. In fact, a sequence (a multiset, a set) of values taken from *V* can be seen as an element of the free monoid *V** (the commutative monoid, the idempotent, and commutative monoid, respectively). The join operation in *V** is written by a comma operator "**,**" and induces the neighborhood of each element: let *E* be a monoidal collection, then the element *y* in *E* is the neighbor of the element *x* iff *E*=*u*, *x*, *y*, *v* for some *u* and *v*. This definition induces the following topology:

- For sets (type set), each element in the set is a neighbor of any other element (because of the commutativity, the term describing a set can be reordered following any order and they are all distinct because of the idempotent property).
- For multisets (type bag), each element is also a neighbor of any other one (however, the elements are not required to be distinct as in a set).
- For sequences (type seq), the topology is the expected one — an element not at one end has a neighbor at its right.

The comma operator is overloaded in MGS and can be used to build any monoidal collection (the type of the arguments disambiguates the collection built). So, the expression 1,1+1, 2+1,():set builds the set with the three elements 1, 2 and 3, while expression 1,1+1,2+1,():seq makes a sequence *s* with the same three elements. The comma operator is overloaded such that if *x* and *y* are not monoidal collections, then x,*y* builds a sequence of two elements. So, the expression 1,1+1,2+1 evaluates to the sequence *s* as well.

Group-Based Data Field

Group-based data fields (GBF for short) are used to define organizations with *uniform* neighborhood. A GBF is an extension of the notion of array, where the elements are indexed by the elements of an abelian group, called the *shape* of the GBF (Giavitto et al., 1995; Michel 1996; Giavitto 2001). For example:

```
gbf Grid2 = < north, east >
```

defines a GBF collection type called Grid2, corresponding to the von Neumann neighborhood in a classical array (a cell above, below, left, or right, but not diagonal). The two names north and east refer to the directions that can be followed to reach the neighbors of an element. These directions are the *generators* of the underlying group structure. The inverse of the generators can also be followed to reach a neighbor. The right-hand side (r.h.s.) of the GBF definition gives a finite presentation of the group structure. The list of the generators can be completed by giving equations that constraint the displacements in the shape:

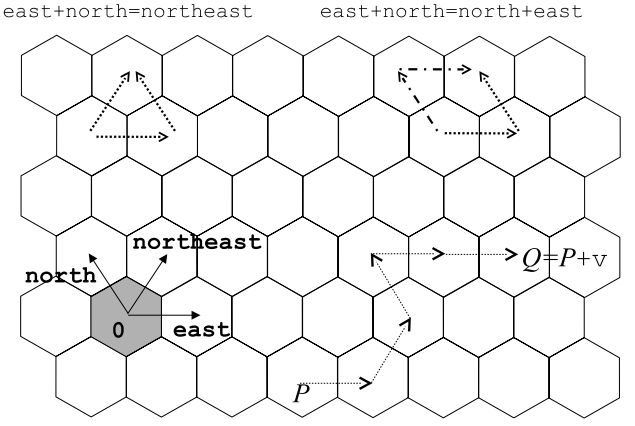
```
gbf Hex = <east, north, northeast; east+north = northeast>
```

defines a hexagonal lattice that tiles the plane, as shown in Figure 2.

In this diagram, an hexagonal cell represents a group element and neighbors' elements share a common edge. Each cell has six neighbors (following the three generators and their inverses). The equation east+north = northeast specifies that a move following northeast is the same as a move following the east direction followed by a move following the north direction. This representation can be easily generalized to visualize the topology of any GBF of type T by a graph. The result is the *Cayley graph* of the presentation of T: each vertex in the Cayley graph is an element of the group, and vertices *x* and *y* are linked if there is a generator *g* in the presentation such that *x* + *g* = *y*. This representation enables a dictionary between graph theoretic notions and group concepts.

A word (a formal sum of the group generators) is a path in the Cayley graph. Path composition corresponds to group addition, and equation *P*+v = *Q*, where *P* and *Q* are given, always has a solution: two cells in the graph are always connected. Each cell *c* can be named by the words that represent a path starting from the cell 0 and ending in *c*. All of these words represent the same group element. A closed path (a cycle) is a word equal to 0 (the identity of the group operation). There are two kinds of cycles in the graph. The cycles that are present in all groups and corresponding to group laws (intuitively, a backtracking path such as b + a – a – b where a and b are generators). The other closed paths are specific to the group equations. An equation v = w can be rewritten v – w = 0 and, thus, corresponds to a closed path. In the diagram, the closed triangular path on the top left corresponds to the equation of the GBF, and the closed path on the top right corresponds to the commutation of the generators east and north. See Figure 2 for an illustration.

Figure 2. Shapes of a GBF <north, east, northeast; east+north = northeast>



A GBF value of type T is a *partial function* that associates a value to some group elements (the group elements are the positions of the collection and the empty GBF is the everywhere undefined function). A GBF value is simply a labeling of a finite set of positions by some values. The positions that have no values are said to be *undefined*.

Matching a Path

Path patterns are used in the left-hand side (l.h.s) of a rule to match a subcollection to be substituted. We give only a fragment of the grammar of the patterns:

Pat ::= *x* | <undef> | *p*,*p'* | *p* |g> *p* | *p*+ | *p*/*exp* | *p* as *x*

where *p*, *p'* are patterns, g is a GBF generator, *x* ranges over the pattern variables, and *exp* is an expression evaluating to a Boolean value. Informally, a path pattern can be flattened into a sequence of basic filters and repetition specifying a sequence of positions with their associated values. The order of the matched elements can be forgotten to see the result of the matching as a subcollection.

A pattern variable *x* matches exactly one element somewhere in the collection that has a well-defined value. The identifier *x* can be used in the rest of the rule to denote the value of the matched element. More generally, the naming of the values of a subpath is achieved using the construction as. The constant <undef> is used to match an element with an undefined value (i.e., a position in a topological collection with no value). The pattern *p*,*p'* stands for a path beginning like *p* and ending like *p'* (i.e., the last element in path *p* must be a neighbor of the first element in path *p'*). For example, the pattern a,b matches two connected elements referred to hereafter as a and b (i.e., b must be a neighbor of a).

The neighborhood relationship depends on the collection kind and is decomposed in several subrelations in the case of a GBF. The comma operator is then refined in the construction *p* |g> *p'*: the first element of *p'* is the g-neighbor of the last element in path *p*. The pattern *p*+ matches a repetition *p*, ..., *p* of path *p*. Finally, *p*/*exp* matches the path *p* only if *exp* holds.

Here is a more contrived example:

(e/seq(e))+ as S / size(S)<5

selects a subcollection S of less than five elements, each element e of S being a sequence. If this pattern is used against a set, S is a subset; if this pattern is used against a sequence, S is a subsequence (that is, an interval of contiguous elements), etc.

Path Substitution and Transformations

There are several features to control the application of a rule. Rules may have priorities or probabilities of application, they may be guarded and depend on the value of local variables, they can "consume" their arguments, etc. We present only the basic application strategy; see Giavitto and Michel (2001) for more details.

Substitutions of Subcollections

A rule $\alpha \rightarrow \beta$ can be seen as a rule for substituting either a path or a subcollection; a path can be seen as a subcollection by simply forgetting the order of the elements in the path. For example, the rule:

$$(x/x<3)+ \text{ as } S \rightarrow 3,4,5,():set$$

applied to the set 1,2,3,4,5,6,():set returns the set 3,4,5,6,():set because S matches the subset 1,2,():set and is replaced by the set 3,4,5,():set. The final result is computed as (3,4,5,():set)$\cup$(3,4,5,6,():set).

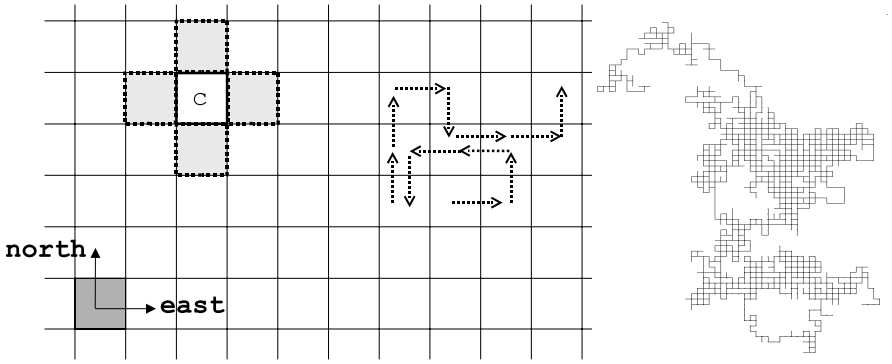
Substitutions of Paths

Because the matched subcollection is also a path — that is, a sequence of elements — the seq type has a special role when appearing in the r.h.s. of a rule. If the r.h.s. evaluates to a sequence, and if this sequence has the same length as the matched path, then the first element of the sequence is used to replace the first element of the matched path, and so on, with the last element in the path replaced by the last element in the sequence. This convention is coherent with the subcollection substitution point of view and simplifies enormously the building of the r.h.s.

For example, suppose that in a GBF we want to model the random walk of a particle x. Then, two neighboring elements, one being x and the other undefined, must exchange their values. This is achieved with only one simple rule:

$$x,<undef> \rightarrow <undef>,x$$

Figure 3. Random walk of a particle on the GBF <north, east>



without the need to mention the precise neighborhood relationships between the two elements. Figure 3 illustrates this process on the GBF <north, east>. This free abelian GBF describes the usual rectangular lattice. Each cell c has four neighbors. Each application of the previous transformation moves the value in a cell to an empty neighbor cell. The path to the right of Figure 3 represents 3,000 random moves on this lattice.

We have mentioned above that the result of replacing a subset by a set is computed using set union. More generally, the insertion of a collection *C* in place of a subcollection *B* depends on the "borders" and on the topology of the involved collections. For example, in a sequence, the subcollection *B* defines in general two borders that are used to glue the ends of collection *C*. The gluing strategy may admit several variations. The programmer can select the appropriate behavior using the rule's attributes.

Transformations

A transformation R is a set of rules:

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

For example, transformation

```
trans M = { x→x+1; }
```

defines a function M. The expression M(c) denotes the application of one transformation step to the collection c. A transformation step consists of the parallel application of the rules (modulo the rule application's features). So, M(c) computes a new collection c' where each element of c is incremented by one.

A transformation step can be easily iterated:

```
M[iter=n](c)
```

denotes the application of n transformation steps, and

```
M[iter=fixpoint](c)
```

denotes the application of M until a fixed point is reached; that is, the result c' satisfies the equation: c'=M(c').

PARADIGMS OF PATTERN FORMATION

In this section, we introduce several fundamental paradigms of pattern formation through some examples and their implementation in MGS. These examples are all fundamental models that have been proposed and discussed in the field of developmental biology. The purpose of this section is not to develop new developmental mechanisms, but to show that these paradigmatic examples can all be easily expressed in the unified framework of topological collection rewriting.

Diffusion and Beyond

Diffusion in a Continuous Setting

Just as thermal gradients cause heat to flow from a warmer area to a colder area, chemical gradients due to variations in chemical concentration cause molecules to move from high concentration to low concentration. This process, due solely to a concentration gradient, is referred to as *diffusion*. The rate of change in concentration with time and space is defined by Fick's law:

$$\frac{\partial C}{\partial t} = D\frac{\partial^2 C}{\partial z^2}$$

where C is the concentration of the chemical (mass/volume), D is some diffusion constant, and z is the spatial variable (to keep the example simple, we suppose that the diffusion occurs in a line).

A forward difference discretization gives

$$C(i,t+\Delta t)=(1-2h)C(i,t)+h\big(C(i-1,t)+C(i+1,t)\big)$$

where $C(i, t)$ represents the concentration at time t of the element of length i. Parameter h depends on the chemical and on the diffusion constant and must be less than 0.5. For the boundary conditions, we assume a source of constant concentration $C(0, t) = C_0$ at one end and a sink that ensures a constant concentration $C(n, t) = 0$ at the other end.

This very simple model can be programmed in MGS in the following way. We first have to define a sequence of elements representing a concentration in a line. This is simply a GBF with only one generator:

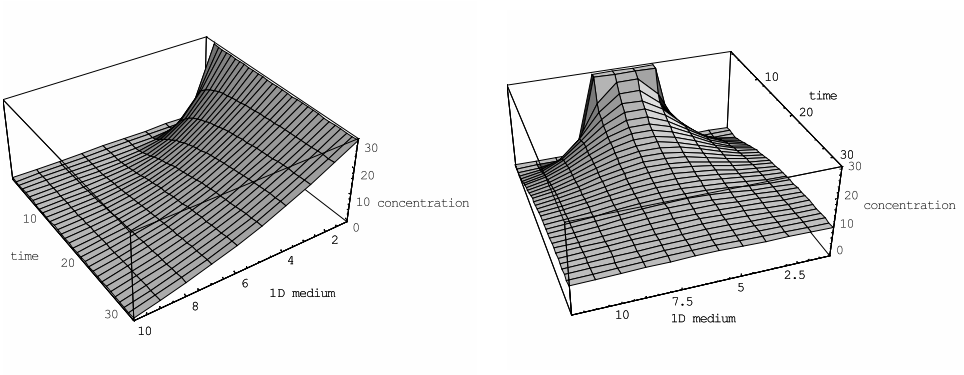
```
gbf Line = <right, left; right+left = 0>
```

The generator left is simply an alias for the inverse of right. These names can be used in a transformation to access the left and right neighbors of an element matched by a pattern variable in a Line. The position (a group element) of the element matched by a pattern variable x is simply denoted by pos(x). Beware that pos, left and right are not functions but special forms that have a meaning only within a transformation. These forms accept as argument only a pattern variable referring to one element in the collection. The transformation that makes the concentration to evolve can be simply written as:

```
trans diffuse[h, C0, n] = {
    x / pos(x) == (0*|right>) → C0;
    x / pos(x) == ((n-1)*|right>) → 0;
    x → (1-2*h)*x + h*(right(x) + left(x))
}
```

h, C0 and n are additional parameters of the transformation. The first two rules deal with the boundary conditions. We assume that the first element of the discretized line is put at the 0*|right> position (this denotes the identity element in the group of positions). Then, the last element is at position (n-1)*|right>. By default, the rules are applied with a priority corresponding to their order of

Figure 4. Result of the diffusion with parameter h=0.156, n=10,C0=30 and two different boundary conditions



declaration: the first rule is applied whenever it can, then the second rule, and the third is possibly applied on the remaining subcollection. The two first rules specifying the behavior on the boundary take precedence to the last rule that governs the default behavior of the interior points.

The last step is to set the initial state S0 of the line. The operator |right> can be used to build this initial collection:

```
S0:=C0 |right> 0 |right> ... |right> 0
```

The evolution from 100 steps is then evaluated by the expression

```
diffuse[h, C0, n, iter=100](S0)
```

The results are visualized in Figure 4. The diagram at the left illustrates the diffusion process with parameter h=0.156, n=10,C0=30,and for 90 time step. The same process but with a different initial state and boundary conditions (no source or sinc) is illustrated on the right.

Diffusion at the Molecular Level

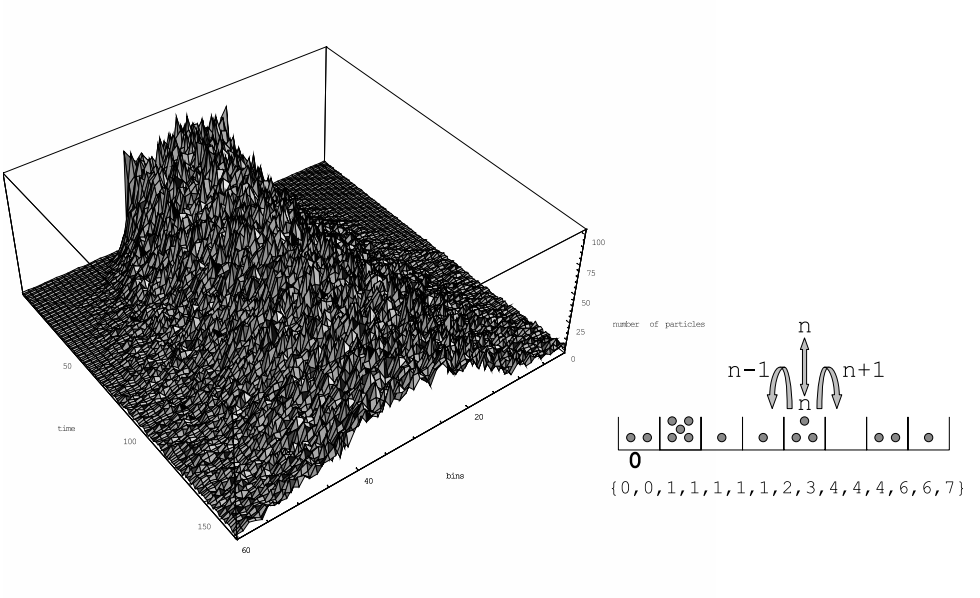
The previous example is very simple but still shows the ability of MGS to handle a continuous model. It is easy to extend this process to a surface or a volume instead of a line.

We now want to take the same system but focus on the level of molecules. The line is still discretized as a sequence of small boxes, indexed by a natural integer, and each containing zero or many molecules. At each time step, a molecule can choose to stay in the same box or to jump to a neighboring box with the same probability (see Figure 5). The state of a molecule is the index of the box in which it resides. The entire state of the system is then represented as a multiset of indices. The evolution of the system can then be specified as a transformation with three rules:

```
trans diffuseM = {
    n → n-1
    n → n
    n → n+1
}
```

The first rule specifies the behavior of a particle that jumps to the box at the left, the second rule corresponds to a particle that stays in the same box, and the last rule defines a particle jumping to the right. Figure 5 illustrates this approach and plots the result of the discrete diffusion of 1,500 particles on a

Figure 5. Right diagram: principle of the particle diffusion model. Left diagram: result of a simulation



sequence of 60 bins during 160 time steps. The diffusion is limited on the boundary (no flow, which is achieved by adding two additional rules to handle the behavior at the boundaries). In the initial states, all particles are randomly distributed in the middle third of the linear media (compare with the right side of Figure 4).

This example shows that even if a multiset has very little organization, it can be used to take geometric information into account. This model can also be extended to diffusion in a surface or a volume, and for arbitrary geometry. The idea is to discretize the medium in a set of bins and to represent the state of the system as a multiset of bins.

Boundary Growth

In this subsection, we focus on variations of cellular automata (von Neumann, 1966) used to model several growth processes.

The Growth of a Snowflake

This cellular automaton idealizes the formation of a snowflake (Wolfram, 2002). Black cells represent regions of solid ice, and white cells represent regions of liquid or gas. The molecules in a snowflake lie on a simple hexagonal grid. Whenever a piece of ice is added to the snowflake, a little heat is released, which then tends to inhibit the addition of further pieces of ice nearby. The corresponding evolution rule is very simple: a cell becomes black whenever exactly one of its neighbors was black in the previous step.

```
Trans Snowflake = {
    0 as x / neighborsfold(+, 0, x)==1 → 1
}
```

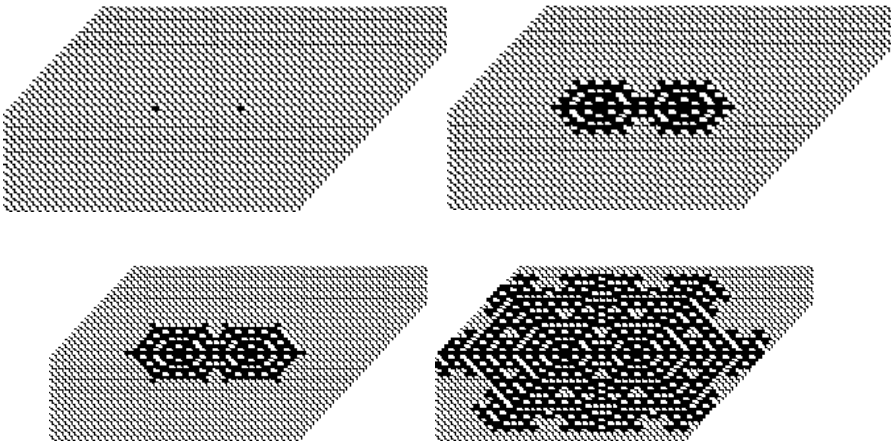
A black cell has the value 1 and a white cell has the value 0. A 0 is turned into a 1 only if the sum of its neighbors is one. The sum of the neighbors is computed using the neighborsfold operator that iterates an accumulating function over the neighbors:

```
neighborsfold(f, zero, x)
```

computes

$$f(x_1, f(x_2, \ldots, f(x_n, \text{zero})\ldots)$$

Figure 6. The growth of a snowflake



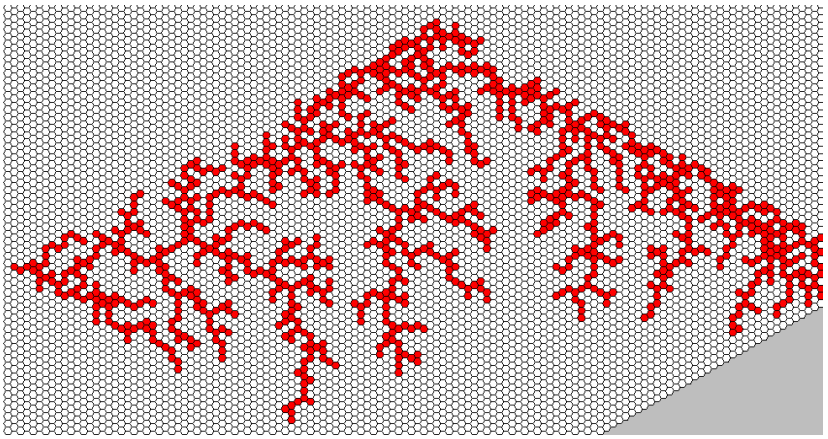
where the x_i are the neighbors of the element x. This operator can be used only within a transformation, and its last argument must be a pattern variable introduced in the left-hand side of the rule. Four steps of the evolution are pictured in Figure 6 (in the initial state, two black cells represent the ice).

Diffusion-Limited Aggregation

Diffusion-Limited Aggregation, or DLA, is a simple process of cluster formation by particles diffusing through a medium that jostles the particles as they move. When particles have the possibility to attract each other and stick together, they may form aggregates. The aggregates may grow as long as there are particles moving around. During the diffusion of a particle it is more likely that it attaches to the outer regions than to the inner ones of the cluster. Thus, a fractal shape occurs like that of corals or trees.

In the following implementation, a value 0 means a diffusing particle, while a value n greater than 1 means a particle fixed to a static cluster for n time steps. Then, the corresponding transformation is straightforward:

Figure 7. A DLA growing process on an hexagonal grid



```
trans DLA = {
    aggregation  =  0,n/n>0 → 1,n
    diffusion =     0,<undef> → <undef>,0
    timecount =     n / n > 0 → n+1
}
```

The rule aggregation specifies that a moving particle that comes near to a cluster will become part of this cluster. The rule diffusion defines the random move of a particle: the particle occupies a free neighboring cell and empties its current occupied cell. The last rule updates the age of a particle stuck into a cluster. The result is illustrated in Figure 7. In this simulation, particles are constantly created on the right, and the initial static particle is completely on the left. Particles are constrained to move on a bounded rhomboidal domain. This explains the asymmetries of the figure.

Phenomenological Sketch of the Growth of a Tumor

This model illustrates the growth of a tumor. It is inspired from a model initially proposed in Wilensky (1998). We start by modeling a set of cells and the mechanical forces between them. Then we add a growing process by giving two different behaviors to the cells. This model gives a formal example of an interacting set of entities localized in a 3-D space, such that the interactions both depend on the position of the entities and make these positions evolve.

The Mechanical Model of the Cells

Each cell exerts a spring-like force to its neighbors. The resulting forces induce cell movements. To keep the model simple, we assume an Aristotelian mechanical physics; that is, the velocity of the cell is proportional to the force exercised. Although this is not compatible with Newtonian physics (the acceleration is actually proportional to the force), the final state (the position of the cells at the equilibrium) is the same and avoids the handling of the acceleration variables.

We represent each cell as a point in $\mathbf{R}^3$ (e.g., assimilated to its mass center) and with some velocity:

```
record Point = {x, y, z}
record Cell = Point + {vx, vy, vz, l, age}
```

These two statements define two new record types named Point and Cell. A Point is a record with the fields x, y, and z (for recording the position of a cell). The Cell type is a subtype of Point having, in addition, the fields vx, vy, and vz (for recording the velocity of a cell) and the fields l and age that record the radius and the age of the cell.

The interaction force between two cells is computed by the function interaction:

```
fun interaction(ref, src) =
    let X = ref.x – src.x
    and Y = ref.y – src.y
    and Z = ref.z – src.z
    and L = ref.l + src.l
    in let dist = sqrt(X*X + Y*Y + Z*Z)
    in let force = (L - dist)/dist
    in {fx=X*force, fy=Y*force, fz=Z*force}
```

The result is a record with fields fx, fy, and fz, which represents the coordinates of the force vector exercised on the cell ref by the cell src. This force goes to infinity when the two interacting cells become closer, it vanishes when the cells are separated by their natural diameter, and becomes asymptotically proportional to the distance between the cells when this distance increases.

A transformation is used to iterate over the cell and to compute the resulting forces:

```
trans Meca = {
    e:Cell  →
    let tf = neighborsfold(sum(e), {fx=0,fy=0,fz=0}, e)
    in e + { x = e.x + epsilon*e.vx,
             y = e.y + epsilon*e.vy,
             z = e.z + epsilon*e.vz,
             vx = tf.fx,
             vy = tf.fy,
             vz = tf.fz }
}
```

The pattern e:Cell is equivalent to e/Cell(e) and selects one element of type Cell. The operator + that appears in the body of the let is overloaded. In addition to the standard numeric addition, it denotes the asymmetric merge of two records. If r and s are two records, then the record r+s contains all of the fields present in r and s. The value of the field f of the record r+s is the value of s.f if f exists in s and else r.f. The net effect of the expression in the body of the let is a record similar to e where the position and the velocity have been updated.

The operator neighborsfold iterates a binary function over the neighbors of an element to compute the total force tf exercised on the cell matched by e. Function sum(e) computes the interaction between e and a neighbor cell and accumulates the result:

```
fun sum(e,s,acc) = addv(acc, interaction(e, s))
fun addv(u,v) = {fx=u.fx+v.fx, fy=u.fy+v.fy, fz=u.fz+v.fz}
```

Note that the function sum is curryed and partially applied in the application of neighborsfold. The operator neighborsfold is similar to the *fold* in functional languages (where it iterates over lists or other algebraic data types) and the use of a curryfied function as the functional argument of the fold is a well-known and heavily used programming pattern (Sheard et al., 1993).

The neighborhood of a cell is computed dynamically using a Delaunay graph built from the cell positions: for a set S of points in the $\mathbf{R}^d$, the Delaunay graph is the unique triangulation of S such that no point in S is inside the circumcircle of any triangle. At each time step, this neighborhood can change due to the cell movements. In MGS, the Delaunay collection type is a type of constructor corresponding to the building of a collection with a neighborhood computed from the position of the elements in a d-dimensional space. A

Delaunay collection type is specified by giving the function that extracts the sequence of coordinates from an element of the collection:

```
delaunay(3) D3 =
\p.if Point(p) then p.x, p.y, p.z
   else error(«Bad element type for a D3») fi
```

The parameter 3 after the keyword delaunay indicates that the elements of this collection type correspond to points in $\mathbf{R}^3$. The notation \x.e is the syntax used for the lambda-expression $\lambda x.e$.

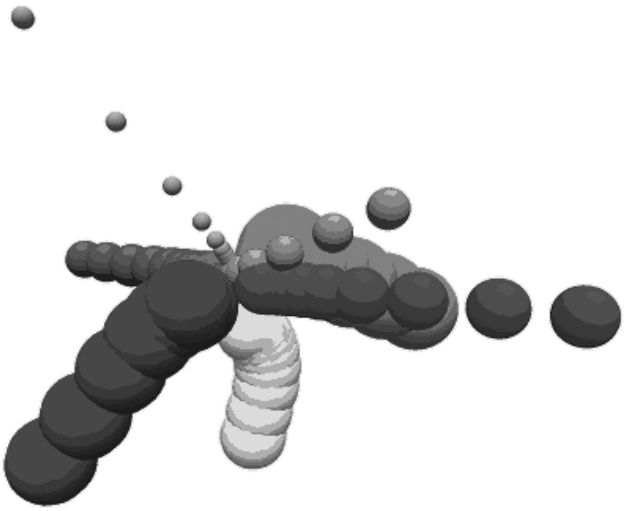
Figure 7 illustrates the trajectory of seven cells computed by iterating the transformation Meca over a collection D3.

The Behavioral Model of the Cells

A tumor consists of stem and transitory cells:

- A transitory cell moves, subject to the forces exercised by other cells.
- A transitory cell may divide at age tc_d with a probability tc_p.
- A transitory cell with an age greater than tc_a eventually dies with a probability tc_d.

Figure 7. A trajectory of seven cells attracted by a spring-like force by the neighbors



- A stem cell is fixed (e.g. anchored in the extra-cellular matrix).
- A stem cell can divide either asymmetrically or symmetrically at some age sc_d. In either case, one of the two daughter cells remains a stem cell, replacing its parent. In asymmetric mitosis, the other daughter is a transitory cell. In symmetric mitosis, the other daughter is a stem cell that is allowed to move for sc_w time before anchoring and being static. The probability of an asymmetric mitosis is sc_p.

These behaviors are specified by the transformation Grow. To select the appropriate cell in the left-hand side of a rule, we introduce the types StemCell and TransitoryCell that are distinguished only by the presence (or the absence) of the field stem.

```
record StemCell = Cell + {stem=true}
record TransitoryCell = Cell + {~stem}
```

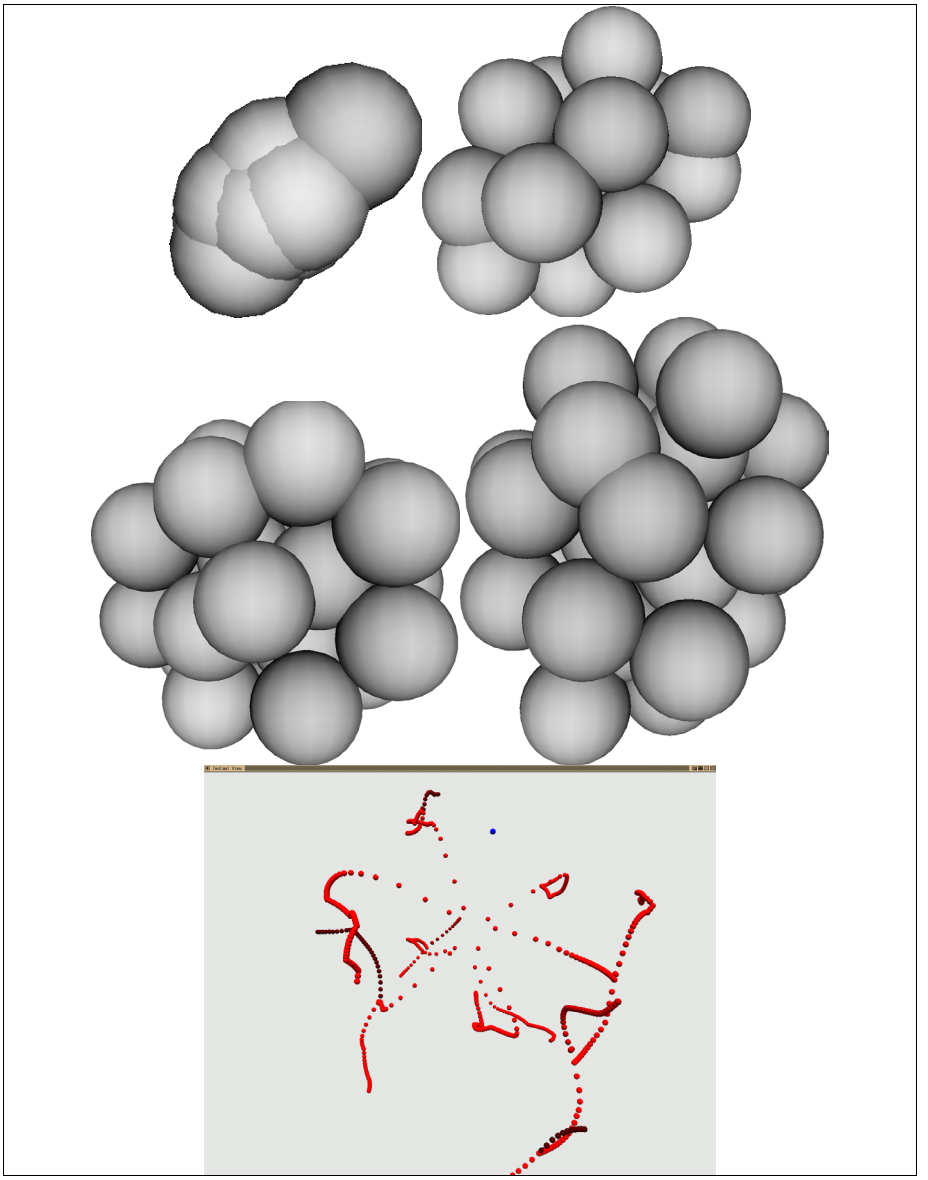
We represent a young stem cell allowed to move as a transitory cell with a negative age. When such kind of cells reaches the age -1, then they transform themselves into static stem cells. Then the definition of Grow can be:

```
trans Grow = {
    c:TransitoryCell / c.age == -1
    →       c+{stem=true, age=0};
    c:TransitoryCell / (c.age > tc_a) & (rand() < tc_d)
    →       <undef>;
    c:TransitoryCell / (c.age == tc_d) & (rand() < tc_p)
    →       r, r+{x=noise(c.x), y=noise(c.y), z=noise(c.z),
             age=0};
    c:StemCell / c.age == sc_d
    →       c, { x=noise(c.x), y=noise(c.y), z=noise(c.z),
             vx=0, vy=0, vz=0, l=c.l,
                    age = if rand()<sc_p then -sc_w else 0 fi
              }
    c:Cell
    →       c+{age=c.age+1};
}
```

The pseudo-function rand returns a random number between 0 and 1. The function noise perturbs its argument: fun noise(x) = x + rand(). So when a cell

divides, one of the two daughter cells inherits the position of the mother cell and the other is a little away. The mechanical interaction then moves the cell away. To restrict the mechanical effects to the transitory cells, it is enough to match

Figure 8. Growth of a tumor (See "Phenomenological Sketch of the Growth of a Tumor" for further explanations)



TransitoryCell instead of Cell in the transformation Meca: the left-hand side of the rule becomes e:TransitoryCell.

Figure 8 illustrates some states in the tumor progression. The first four images are drawings of the tumor at different time moments (cells have divided and rearranged to satisfy the mechanical constraints). The last image (at the bottom side) corresponds to the trajectory of the mass center of each cell, before the first mitosis of a stem cell. The division of a transitory cell (visible as forks in the trajectories) induces a change in the mechanical constraints and a corresponding change in the trajectory.

This kind of simulation is phenomenological because the behavior of each cell is roughly modeled, without taking into account the chemical and genetic processes. However, this kind of simulation can be used to estimate the propagation of the tumor, to evaluate the ratio between transitory and stem cells, and to evaluate the impact of various therapeutic strategies, such as cell division or mobility inhibitors. Most of the chemotherapy drugs known as M- and S-poisons inhibit cell division and impact mainly the transitory cells. The problem is that the stem cells (also known as clonogenic cells) maintain the tumor and propagate its metastases. Other possible approaches try to lower the cell mobility to reduce the tumor propagation.

DISCUSSION AND CONCLUSIONS

Summary

The MGS programming language is largely inspired by the dynamical system perspective on biological development. From this point of view, biological development exhibits a dynamical structure, or a variable phase space, that must be computed jointly with the current state of the system. While it makes the classical modeling and the simulation of such systems very difficult, their description is still usually easily achieved by a set of local evolution rules specifying the transformation of a subsystem. However, the partition of the system into subsystems evolves in the course of time and the conditions of a transformation application can be complex. These considerations lead to the development of a rule-based programming paradigm. This programming paradigm is characterized by the repeated, localized transformations of a shared data object. The transformations are described by *rules* that separate the description of the subobject to be replaced (the *pattern*) from the calculation of the replacement.

Optionally, rules can have further conditions that restrict their applicability and the transformations are controlled by explicit or implicit *strategies*. When the data object is a term, we retrieve the notion of rewrite systems. MGS extends this approach by considering objects structured by neighborhood relationships. The topological approach unifies several models of computations, at least to provide a single syntax that can be consistently used to allow the merging of these formalisms for programming purposes. A unifying theoretical framework can be developed (Giavitto & Michel, 2001; Giavitto & Michel, 2002), based on the notion of *chain complex* developed in algebraic combinatorial topology.

The resulting programming style is an effective framework for the modeling and the simulation of various developmental processes, as shown in the previous section. All of the examples in this chapter have been processed using the MGS interpreter. Theoretical articles, documentations, and various MGS software products are freely available at: *http://mgs.lami.univ-evry.fr*.

Related Works

Transformation on multisets is reminiscent of multiset rewriting (or rewriting of terms). This is the main computational device of Gamma (Banâtre & Le Metayer, 1986; Banâtre et al., 1987, 2001). The CHemical Abstract Machine (CHAM) extends these ideas with a focus on the semantics of nondeterministic processes (Berry & Boudol, 1992). The CHAM introduces a mechanism to isolate some parts of the chemical solution. This idea has been seriously taken into account in the context of P systems. P systems (Păun, 2001) are a new distributed parallel computing model based on the notion of a membrane structure. A membrane structure is a nesting of cells represented by a Venn diagram without intersection and with a unique superset: the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations: an object can evolve into another object, can pass through a membrane, or dissolve its enclosing membrane. As for Gamma, the computation is finished when no object can further evolve. By using nested multisets, MGS is able to emulate more or less the notion of P systems. In addition, patterns like the iteration + go beyond what is possible to specify in the l.h.s. of a Gamma rule.

Lindenmayer systems (Lindenmayer, 1968) have long been used in the modeling of $(DS)^2$ (especially in the modeling of plant growth). They loosely correspond to transformations on sequences or string rewriting (they also correspond to tree rewriting because some standard features make arbitrary

trees that are particularly simple to code). Obviously, L systems are dedicated to the handling of linear and tree-like structures.

Strong links exist between GBF and cellular automata (CA), especially considering the work of Z. Róka, which has studied CA on Cayley graphs (1994). However, our own work focuses on the construction of Cayley graphs as the shape of a data structure, and we have developed an operator algebra and rewriting notions on this new data type. This is not in the line of Z. Róka, which focuses on synchronization problems and establishes complexity results in the framework of CA.

In the domain of biological process modeling, an increasing research effort is devoted to the design of a simulation platform at the cellular level. The current projects are mainly based on the modeling of the metabolic activities through differential equations (ODEs) or partial differential equations (PDEs). They act then as ODE or PDE solvers dedicated to biological processes. For instance, BioDrive (Kyoda et al., 2000) handles signal transduction. E-Cell is dedicated to the design of a minimal set of genes coding basic metabolic functions and includes an evaluation of the corresponding cell functioning through an energetic cost (Tomita et al., 1999; *www.e-cell.org*). V-Cell is one of the most advanced simulation platforms — it handles PDEs and enables the specification of complex geometry (Schaff & Loew, 1999; *www.nrcam.uchc.edu*). These simulators rely on the hypothesis that the chemical activities that occur in the cells are adequately described only by their kinetic equations (as it is the case in a test tube). Consequently, they are unable to model and ignore the dynamic organizations that modify profoundly the reactions, such as compartimentalization, the creation of hyperstructures (Amar et al., 2002), or the channeling in the case of metabolon by the so-called "solid-state metabolism" effect.

Other modeling approaches rely on the multi-agent paradigm to model the various entities and activities that appear in biological processes. The resulting software architecture support fits very well into the description of the domain's ontology (the specification of an ontology of biological entities and activities is a problem in itself considering the vast number of different entities and activities to describe). However, this approach does not bring a solution by itself to the problem of describing the interaction between an arbitrary collection of agents and the spatial organization of the agents. To overcome this problem, some current projects extend the multi-agent paradigm with other approaches, such as cellular automata.

The MGS programming style corresponds to the rule-based programming paradigm. Rule-based programming is currently experiencing a renewed

period of growth with the emergence of new concepts and systems that allow a better understanding and better usability. However, the vast majority of rule-based languages (such as expert systems) are funded on a logical approach (computation is a logical deduction), which is not adequate to describe various biochemical processes. Yet the algorithmic and biological examples given in the two previous sections demonstrate the ability of MGS to express both discrete and (the numerical solution of) continuous models.

Perspectives

The perspectives opened by this preliminary work are numerous. We want to develop several complementary approaches to define new topological collection types. One approach to extend GBF applicability is to consider monoids instead of groups, especially automatic monoids, which exhibit good algorithmic properties. Another direction is to handle general combinatorial spatial structures such as simplicial complexes. At the language level, the study of the topological collections concepts must continue with a finer study of transformation types. Several kinds of restrictions can be considered with regard to these transformations, leading to various kinds of pattern languages and rules. The complexity of matching such patterns has to be investigated. The efficient compilation of an MGS program is a long-term research plan. We have considered in this chapter only one-dimensional paths, but a general *n*-dimensional notion of paths exists and can be used to generalize the substitution mechanisms of MGS. From the applications point of view, we are looking for simulations of more complex developmental processes.

ACKNOWLEDGMENTS

The authors would like to thank M. Gheorghe at the University of Sheffield, R. Paton and G. Malcolm at the University of Liverpool, F. Delaplace at the University of Evry, C. Godin and P. Barbier de Reuille at CIRAD-Montpellier, the members of the epigenomic group at GENOPOLE-Evry, P. Prusinkiewicz at the University of Calgary, and the organizers of the friendly "workshop on membrane computing" series for helpful discussions, biological motivations, fruitful examples, and challenging questions. Further acknowledgments are also due to J. Cohen, A. Spicher, B. Calvez, F. Thonnerieux, C. Kodrnja, and F. Letierce who have contributed in various ways to the MGS software. This research is supported in part by the French National Center for

Scientific Research (CNRS), GENOPOLE-Evry, the national working group GDR ALP and IMPG, and the University of Evry Val d'Essonne.

REFERENCES

Amar, P., Giavitto, J.-L., Michel, O., Norris, V., & 36 other co-authors (2002). Hyperstructures, genome analysis and I-cells. *Acta Biotheoretica*, *50*(4), 357-373.

Banâtre, J.-P. & Le Metayer, D. (1986). A new computational model and its discipline of programming. *Technical Report RR-0566*, Inria.

Banâtre, J.-P., Coutant, A., & Le Metayer, D. (1987). Parallel machines for multiset transformation and their programming style. *Technical Report RR-0759*, Inria.

Banâtre, J.-P., Fradet, P., & Le Metayer, D. (2001). Gamma and the chemical reaction model: Fifteen years after. In Calude, C.S., Păun, Gh., Rozenberg, G., & Salomaa, A. eds. *Multiset processing: Mathematical, computer science, and molecular computing points of view*. Lecture notes in *Computer Science*, 2235, Berlin: Springer, 17-31.

Berry, G. & Boudol, G. (1992). The chemical abstract machine. *Theoretical Computer Science*, 96: 217-248.

Bournez, O., Kirchner, H., Côme, G.-M., Conraud, V., & Ibanescu, L. (2003). A rule-based approach for automated generation of kinetic chemical mechanisms. In R. Nieuwenhuis, ed. *14th Int. Conf. on Rewriting Techniques and Applications*. Lecture notes in *Computer Science*, 2706, Berlin: Springer, 30-45.

Buneman, P., Naqvi, S., Val Tannen, B., & Wong, L. (1995). Principles of programming with complex objects and collection types. *Theoretical Computer Science*, *149* (1), 3-48.

Dershowitz, N. & Jouannaud, J.-P. (1990). Rewrite systems. *Handbook of Theoretical Computer Science*, Vol. B. 244-320.

Dershowitz, N. (1993). A taste of rewrite systems. Lecture notes in *Computer Science*, 693, Berlin: Springer, 199-228.

Dittrich, P., Ziegler, J., & Banzhaf, W. (2001). Artificial chemistries – a review. *Artificial Life*, *7* (3), 225-275.

Eigen, M. & Schuster, P. (1979). *The Hypercycle: A Principle of Natural Self-Organization*. Berlin: Springer.

Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., & Kemal Sönmez, M. (2002). Pathway logic: Symbolic analysis of biological

signaling. In *Pacific Symposium on Biocomputing PSB 2002*, 400-412.
Fontana, W. & Buss, L. (1994). "The arrival of the fittest": Toward a theory of biological organization. *Bulletin of Mathematical Biology*. 56, 1-64.
Fisher, M., Malcolm, G., & Paton, R. (2000). Spatio-logical processes in intracellular signalling. *BioSystems*, 55, 83-92.
Giavitto, J.-L. (2001). Declarative definition of group indexed data structures and approximation of their domains. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming PPDP01*. ACM Press, 150-161.
Giavitto, J.-L. (2003). Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In Nieuwenhuis, R. ed. *14th International Conference on Rewriting Techniques and Applications RTA'03*. Lecture notes in *Computer Science*, 2706, Berlin: Springer, 208-233.
Giavitto, J.-L. & Michel, O. (2001). MGS: A programming language for the transformations of topological collections. *LaMI Technical Report 61-2001*. University of Évry Val d'Essonne, France.
Giavitto, J.-L. & Michel, O. (2002). The topological structures of membrane computing. *Fundamenta Informaticae*, *49* (1-3), 107-129.
Giavitto, J.-L. & Michel, O. (2003). Modeling the topological organization of cellular processes. *BioSystems*, *70*(2), 149-163.
Giavitto, J.-L., Malcolm, G., & Michel, O. (2004). Rewriting systems and the modelling of biological systems. *Comparative and Functional Genomics*, *5*(1), 95-99.
Giavitto, J.-L., Michel, O., & Sansonnet, J.-P. (1995). Group based fields. In *Parallel Symbolic Languages and Systems PSLS95*. Lecture notes in *Computer Science*, 1068, Berlin: Springer, 209-215.
Hammel, M. & Prusinkiewicz, P. (1996). Visualization of developmental processes by extrusion in space-time. *Proceedings of Graphics Interface '96*, 246-258.
Harper, J.L., Rosen, B.R., & White, J. (1986). *The Growth and Form of Modular Organism.* London: The Royal Society.
Head, T. (1987). Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, *49* (6), 737-759.
Head, T. (1992). Splicing schemes and DNA. In *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*. Berlin: Springer, 371-383. Also appears in (1992). *Nanobiology*, 1, 335-342.

Hung, J.Y., Gao, W., & Hung, J.C. (1993). Variable structure control: A survey. *IEEE Transactions on Industrial Electronics, 40* (1), 2-22.

Itkis, Y. (1976). *Control Systems of Variable Structure*. New York: Wiley.

Kaufman, S. (1995). *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford: Oxford University Press.

Keller, E.F. (1995). *Refiguring Life: Metaphors of Twentieth-Century Biology*. New York: Columbia University Press.

Kephart, J. & Chess., D. (2003). The vision of autonomic computing. *IEEE Computer Magazine, 36*(1) 41-50.

Kyoda, K.M., Muraki, M., & Kitano, H. (2000). Construction of a generalized simulator for multi-cellular organisms and its application to Smad signal transduction. In *Fifth Pacific Symposium on Biocomputing PSB 2000*, 314-325.

Lincoln, P. (2003). Symbolic systems biology. In Nieuwenhuis, R. ed. *14th International Conference on Rewriting Techniques and Applications RTA'03*. Lecture notes in *Computer Science*, 2706, Berlin: Springer, 1.

Lindenmayer, A. (1968). Mathematical models for cellular interaction in development, parts I and II. *Journal of Theoretical Biology*, 18, 280-299 and 300-315.

Maynard-Smith, J. (1999). *Shaping Life: Genes, Embryos and Evolution*. New Haven, CT: Yale University Press.

Meinhardt, H. (1982). *Models of Biological Pattern Formation*. New York: Academic Press.

Michel, O. (1996). Design and implementation of 81/2, a declarative data-parallel language. *Computer Language, 22*(2/3), 165-179.

Mjolsness, E., Sharp, D.H., & Reinitz, J. (1991). A connectionist model of development. *Journal of Theoretical Biology, 152* (4), 429-454.

Munkres, J.R. (1993). *Elements of Algebraic Topology*. Addison-Wesley.

Parashar, M. & Hariri, S., eds. (2003). Autonomic applications workshop. Taj Krishna, Hyderabad, India. Special issue of *Cluster Computing, The Journal of Networks, Software Tools and Applications* (2004).

Paton, R. ed. (1994). *Computing with Biological Metaphors*. London, New York: Chapman & Hall.

Păun, Gh. (2001). From cells to computers: Computing with membranes (P systems). *BioSystems, 59*(3), 139-58.

Prusinkiewicz, P. (1998). Modeling of spatial structure and development of plants: A review. *Scientia Horticulturae*, 74, 113-149.

Prusinkiewicz, P. (1999). A look at the visual modeling of plants using L-systems. *Agronomie*, 19, 211-224.

Prusinkiewicz, P. & Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants*. Berlin: Springer.

Róka, Z. (1994). One-way cellular automata on Cayley graphs. *Theoretical Computer Science, 132* (1–2), 259-290.

Schaff, J. & Loew, L.M. (1999). The virtual cell. In *Fourth Pacific Symposium on Biocomputing PSB 1999*, 4, 228-239.

Sheard, T. & Fegaras, L. (1993). A fold for all seasons. *Proceedings of the 6th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture FPCA'93*, ACM Press, 233-242.

Stengers, I. (1988). D'une science à l'autre. *Les Concepts Nomades*. Paris: Le Seuil.

Stevens, P.S. (1974). *Patterns in Nature*. Boston: Little, Brown and Co.

Thompson, D'Arcy W. (1942). *On Growth and Form*. Cambridge: University Press.

Tomita, M., Hashimoto, M., Takahashi, M., Shimizu, T.-S., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Yugi, K., Venter, J.-C., & Hutchison, C.-A., (1999). E-CELL: Software environment for whole cell simulation. *Bioinformatics, 15* (1),72-84.

Turing, A.M. (1952). The chemical basis of morphogenesis. *Philosophical Transactions*. Royal Society of London, Series B: Biological Sciences, (237), 37-72.

Varela, F.J. (1979). *Principle of Biological Autonomy*. New York: McGraw-Hill/Appleton and Lange.

von Neumann, J. (1966). *Theory of Self-Reproducing Automata*. Urbana, IL: University of Illinois Press.

Wolfram, S. (2002). *A New Kind of Science*. Champaign, IL: Wolfram Media, Inc.

Wolpert, L., Beddington, R., Lawrence, P., Meyerowitz, E., Smith, J., & Jessell, T.M. (2002). *Principles of Development* (2nd ed.). Oxford: Oxford University Press.

Wilcox, M., Mitchison, G.J., & Smith, R.J. (1973). Pattern formation in the blue-green alga, *Anabaena*. I. Basic mechanisms. *Journal of Cell Science*, 12, 707-723.

Wilensky, U. (1998). NetLogo tumor model. Contributed by Gershom Zajicek M.D., Prof. of Experimental Medicine and Cancer Research at The Hebrew University-Hadassah Medical School, Jerusalem. *http://ccl.northwestern.edu/netlogo/models/Tumor*. Center for Connected

Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

ENDNOTE

1 As stated by d'Arcy W. Thompson (1942): "We might call the form of an organism an event in space-time, and not merely a configuration in space."