

Declarative simulation of dynamical systems: the $8\frac{1}{2}$ programming language and its application to the simulation of genetic networks

Jean-Louis Giavitto*, Olivier Michel, Franck Delaplace

LaMI u.m.r. 8042 du CNRS, Université d'Evry Val d'Essone, 91025 Evry Cedex, France

Abstract

A major part of biological processes can be modeled as dynamical systems (DS), that is, as a time-varying state. In this article, we advocate a declarative approach for prototyping the simulation of DS. We introduce the concepts of collection, stream and fabric. A fabric is a multi-dimensional object that represents the successive values of a structured set of variables. A declarative programming language, called $8\frac{1}{2}$ has been developed to support the concept of fabrics. Several examples of working $8\frac{1}{2}$ programs are given to illustrate the relevance of the fabric data structure for simulation applications and to show how recursive fabric definitions can be easily used to model various biological phenomena in a natural way (a resolution of PDE, a simulation in artificial life, the Turing diffusion-reaction process and various examples of genetic networks). In the conclusion, we recapitulate several lessons we have learned from the $8\frac{1}{2}$ project.
© 2002 Elsevier Science Ireland Ltd. All rights reserved.

Keywords: Declarative programming languages; Simulation of dynamical systems; Biological processes; Stream; Collection

1. Introduction

1.1. The simulation of dynamical systems

Dynamical systems (DS) are an abstract framework used to model phenomena that occur in space and time. The system is characterized by 'observable', called the variables of the system, which are linked by some relations. The value of the variables evolves with the time. A variable can take a scalar value (like a real) or be of a more

complex type like the variation of a simpler value on a spatial domain. An example of such a complex type is the temperature on each point of a room or the velocity of a fluid in a pipe. This last kind of variable is called a field. The set of the values of the variables that describe the system constitutes its state. The state of a system is its observation at a given instant. The sequence of state changes is called the trajectory of the system.

Intuitively, a DS is a formal way to describe how a point (the state of the system) moves in the phase space (the space of all possible states of the system). It gives a rule telling us where the point should go next from its current location (the evolution function).

* Corresponding author.

E-mail address: giavitto@lami.univ-evry.fr (J.-L. Giavitto).

Table 1

Some formalisms used to specify a DS following the discrete or continuous nature of space, time and value

C: continue, D: discrete	PDE	ODE	Iterated equations	Cellular automata
Space	C	D	D	D
Time	C	C	D	D
State	C	C	C	D

PDE, partial differential equation; ODE, ordinary differential equation.

There exists several formalisms used to describe a DS: ordinary differential equations (ODE), partial differential equations (PDE), iterated equations (finite set of coupled difference equations), cellular automata, etc. In the [Table 1](#), the discrete or continuous nature of the time, the space and the value, is used to classify some DS specification formalisms.

The study of these kinds of models can be found in all scientific domains and make often use of digital simulations. As a matter of fact, it is sometimes too difficult, too expensive or simply impossible to make real experiments (e.g. for ethical reasons). The US ‘Grand Challenge’ initiative to develop the hardware and software architectures needed to reach the tera-ops, outlines that numerical experiments, now mandatory in all scientific domains, is possible only if all the computing resources are easily available, see [NSF \(1991\)](#).

From this point of view, the expressiveness of a simulation language is at least as important as its efficiency. Nowadays the data structures and the algorithms used are indeed more and more sophisticated. The lack of expressive power becomes then an obstacle to the development of new simulation programs. If an imperative language like FORTRAN-77 is used to develop a DS simulation, most of the time dedicated to programming will be spent in the burden of representation of the objects of the simulation, memory management, management of the logical time, management of the scheduling of the activities of the objects of the simulation, ... A high-level DS simulation language must then offer well fitted dedicated concepts and resources to relieve the programmer from making many low-level implementation decisions and to concentrate the complexity of the algorithms in dedicated data and control struc-

tures. Certainly, this implies some loss of run-time performance but in return for programming convenience. How much loss we can tolerate and what do we get in exchange must be carefully evaluated.

1.2. The 8½ language for DS simulations

These considerations have driven the 8½ project. The goal of this long time effort is to design a high-level parallel language for the simulation of DS, cf. [Michel et al. \(1994\)](#) and [Michel and Giavitto \(1998b\)](#). For instance, the various formalisms¹ cited in [Table 1](#) are naturally expressed in 8½. In this paper, we focus on a general presentation of 8½ towards the simulation of some biological DS. Issues like parallelism or implementation are eluded (the reader may refer to [Michel et al., 1994](#); [Michel and Giavitto, 1994](#); [Mahiout and Giavitto, 1994](#); [De Vito and Michel, 1996](#)).

We have naturally chosen a declarative style close to the mathematical formalism used in DS specifications, see [Michel et al. \(1994\)](#) and [Michel and Giavitto \(1998b\)](#). We have designed in this declarative framework a new data structure: the fabric². A fabric represents the trajectory of a DS. It is a temporal sequence (a stream) of collections (a collection is a set of data simultaneously accessible and managed as a whole).

¹ Obviously, PDE and ODE are discretised before their numerical resolution but the numerical schema is directly written as a 8½ program, see for example [Section 3.1](#).

² This data structure has been initially called web because the interleaving between the weft and the warp in threads woven gives an accurate image of the interplay of streams and collections in the recursive definition of a fabric. However, the ambiguity raised by the development of the Internet has motivated the change of name. Both names can be found in our papers.

It is only recently that biological DS have been considered as an application area for the $8\frac{1}{2}$ language. One of our main interest is the systematic development of the simulation of biochemical networks. The examples worked in this paper show that the formalism is very well-fitted for DS whose structure is static. Examples of this kind of system are: genetic networks, predator–prey systems, etc.

However, we share the conclusion drawn by Fontana and Buss (1996) that the modeling of several fundamental biological processes require the capacity of computing the state space jointly with the running state of the process. These applications represented nowadays a new frontier in the modeling of DS and has motivated the beginning of a new project.

1.3. Organization of the paper

The rest of this paper is organized as follow: the next section present the concept of collection, stream and the coupling of the two structure in a fabric. Section 3 gives the example of the resolution of a PDE, and the simulation of an artificial creature whose behavior is triggered by the internal level of some variables. We finish by the classical example of Turing’s model of morphogenesis. Section 4 continues the presentation of $8\frac{1}{2}$ through several example of genetic networks simulation models. The objective is to show how the variation of models are handled by slight changes in the $8\frac{1}{2}$ programs. Section 5 gives some examples of DS with a dynamical structure. In conclusion, we quickly review some of the lessons learned on the $8\frac{1}{2}$ project.

2. Recursive definition of stream, collection and fabrics

Programming language $8\frac{1}{2}$ has a single data structure called a fabric. A fabric is the combination of the concepts of stream and collection. This section describes these three notions.

2.1. The concept of collection in $8\frac{1}{2}$

A collection is a data structure that represents a set of elements as a whole, like in Blleloch and Sabot (1990). Several kinds of aggregation structures exist in programming languages: set in SETL, see Schwartz et al. (1986) and Jayaraman (1992), list in LISP, tuple in SQL, pvar in LISP, cf. TMC (1986) or even finite discrete space in Cellular Automata, see Tofooli (1987). Data-parallelism is naturally expressed in terms of collections introduced in Sipelstein and Blleloch (1991). From the point of view of the parallel implementation, the elements of a collection are distributed over the processing elements (PEs).

Here, we consider collections that are ordered sets of elements³. An element of a collection, also called a point in $8\frac{1}{2}$ is accessed through an index. The expression $T \cdot n$ where T is a collection and n an integer, is a collection with one point; the value of this point is the value of the n th point of T (point numbering begins with 0). If necessary, we implicitly coerce a collection with one point into a scalar and vice-versa through a type inference system described in Giavitto (1992).

Geometric operators change the geometry of a collection, i.e. its shape or structure. The geometry of a collection of scalars is reduced to its cardinal (the number of its points). A collection can also be nested: the value of a point is a collection. The geometry of the collection is the hierarchical structure of point values.

The first geometric operation consists in packing some fabrics together:

$$T = \{a, b\}$$

In the previous definition, a and b are collections resulting in a nested collection T . Elements of a collection may also be named and the result is then a system. Assuming:

$$\text{car} = \{\text{velocity} = 5, \text{consumption} = 10\}$$

the points of this collection can be reached

³ More generally, $8\frac{1}{2}$ collections are multidimensional arrays, fields (functional partial arrays introduced in Lisper (1993)) or GBF (partial arrays whose elements are indexed by an element in a group, investigated in Giavitto et al. (1995)).

uniformly through the dot construct using their label, e.g. car.velocity, or their index: car.0.

The composition operator # concatenates the values and merges the systems:

$$A = \{a, b\}; \quad B = \{c, d\};$$

$$A\#B \Rightarrow \{a, b, c, d\}$$

$$\begin{aligned} \text{ferrari} &= \text{car}\#\{\text{color} = \text{red}\} \\ &\Rightarrow \{\text{velocity} = 5, \text{consumption} = 10, \text{color} \\ &= \text{red}\} \end{aligned}$$

The last geometric operator we will present here is the selection: it allows the selection of some point values to build another collection. For example:

$$\begin{aligned} \text{Source} &= \{a, b, c, d, e\} \\ \text{target} &= \{1, 3, \{0, 4\}\} \\ \text{Source}(\text{target}) &\Rightarrow \{b, d, \{a, e\}\} \end{aligned}$$

The notation Source(target) has to be understood in the following way: a collection can be viewed as a function from $[0..n]$ to some co-domain. Therefore, the dot operation corresponds to function application. If the co-domain is the set of natural numbers, collections can be composed and the following property holds: $\text{Source}(\text{target}.i) = \text{Source}(\text{target}.i)$, mimicking the function composition definition.

Four kinds of function applications can be defined (Table 2).

X means both scalar or collection; p is the arity of the functional parameter f . The first operator is the standard function application. The second type of function applications produces a collection whose elements are the ‘pointwise’ applications

of the function to the elements of the arguments. Then, using a scalar addition, we obtain an addition between collections. Extension is implicit for the basic operators (+, *, etc.) but is explicit for user-defined functions to avoid ambiguities between application and extension (consider the application of the reverse function to a nested collection).

The third type of function applications is the reduction. Reduction of a collection using the binary scalar addition, results in the summation of all the elements of the collection. Any associative binary operation can be used, e.g. a reduction with the min function gives the minimal element of a collection. The scan application mode is similar to the reduction but returns the collection of all partial results. For instance: $+\backslash\{1, 1, 1\} \Rightarrow \{1, 2, 3\}$. See [Blelloch \(1989\)](#) for a complete algorithmic based on scan.

2.2. The concept of stream in 8½

2.2.1. Dealing with infinite sequence of values

LUCID, cf. [Wadge and Ashcroft \(1976\)](#), is one of the first programming languages defining equations between infinite sequences of values. Although 8½ streams are also defined through equations between infinite sequences of values, 8½ streams are very different from those of LUCID. They are tightly linked with the idea of observing a remanent state along time.

A metaphor to explain 8½ streams is the sequence of values of a measuring apparatus. If you observe a measuring apparatus during an experiment run, you can record the successive measure operations on this apparatus, together with their dates. The timed sequence of data is a 8½ stream. At the very beginning, before the start

Table 2

Operator	Signature	Syntax
Application	$(\text{collection}^p \rightarrow X) \times \text{collection}^p \rightarrow X$	$f(c_1, \dots, c_p)$
Extension	$(\text{scalar}^p \rightarrow \text{scalar}) \times \text{collection}^p \rightarrow \text{collection}$	$f(c_1, \dots, c_p)$
Reduction	$(\text{scalar}^2 \rightarrow \text{scalar}) \times \text{collection} \rightarrow \text{scalar}$	$f\backslash c$
Scan	$(\text{scalar}^2 \rightarrow \text{scalar}) \times \text{collection} \rightarrow \text{collection}$	$f\backslash\backslash c$

of the experiment, the initial value of any observable is an undefined value. Then we record the initial value (at time 0 for some observables, later for some others). This value can be read and used to compute other values recorded elsewhere, as long as another observation has not been made.

The time used to label the observation is not the computer physical time, it is the logical time linked to the semantics of the program. The situation is exactly the same between the logical time of a discrete-events simulation and the physical time of the computer that runs the simulation. Therefore, the time to which we refer is a countable set of ‘events’. An event is something meaningful for the simulation, like a change in a value.

2.2.2. *The pace of a stream: ticks, tocks and clocks*

The programming language 8½ is a declarative language, which operates by making descriptive statements about data and relationships between data, rather than by describing how to produce them.

For instance, the definition $C = A + B$ means the value recorded by stream C is always equal to the sum of the values recorded by stream A and B . We assume that the changes of the values are propagated instantaneously. When A (or B) changes, so do C at the same logical instant. Note that C is uninitialized as long as A or B are uninitialized.

Table 3 gives some examples of 8½ streams operations. The first line gives the instants of the logical clock, which counts the events in the program. The instants of this clock are called a tick (a tick is a column in the table). The dates of the recording of a new observation for a particular observable are called the tock of this stream

(because a large clock is supposed to make ‘tick–tock’). Tocks represent the set of events meaningful for that stream. A tock is a non-empty cell in the table.

You can always observe your measuring apparatus, which gives the result of the last measurement, until a new measure is made. Consequently, at a tick t , the value of a stream is: the last value recorded at tock $t' \leq t$ if t' exists, or the undefined value otherwise. For example, the value of $\$C$ at tick 0 is undefined whilst its value at tick 4 is 3.

2.2.3. *Stream operations*

A scalar constant stream is a stream with only one ‘measurement’ operation, at the beginning of time, to compute the constant value of the stream. A constant n in a 8½ program, really denotes a scalar constant stream.

Constructs like $\text{Clock } n$ denote another kind of constant streams: they are predefined sequences of true values with an infinite number of tocks. The set of tocks depends of the parameter n . They really represent some clocks used to give the beat of some other observations.

Scalar operations are extended to denote element wise application of the operation on the values of the streams.

The delay operator, $\$$, shifts the entire stream to give access, at the current time, to the previous stream’s value. This operator is the only operator that does not act in a point-wise fashion. The tocks of the delayed stream are the tocks of the arguments at the exception of the first one.

The last kind of stream operators are the sampling operators. The most general one is the trigger. It corresponds to the temporal version of

Table 3
Examples of constant streams and stream expressions

	0	1	2	3	4	5	6	7	8	...
1	11									...
1+2	3									...
Clock 2	True		True		true		true		true	...
Assuming A	1		2	3		4	5	6		...
Assuming B		1		2		1		1		...
$C = A + B$		2	3	5		6	6	7	7	...
$\$C$			2	3		5	6	6	7	...

the conditional. The values of ‘ T when B ’ are those of T sampled at the tocks where B takes a true value (see Table 4). A tick t is a tock of ‘ A when B ’ if A and B are both defined and t is a tock of B and the current value of B is true.

$8\frac{1}{2}$ streams present several advantages:

- $8\frac{1}{2}$ streams are manipulated as a whole, using filters, transducers... cf. Arvind and Brock (1983).
- A stream is the ideal implementation for the trajectory of a DS: a temporal sequence of values is represented by a temporal succession of computation and, therefore, can be infinite.
- The tocks of a stream really represent the logical instants where some computation must occur to maintain the relationships stated in the program.
- The $8\frac{1}{2}$ stream algebra verifies the causality assumption; the value of a stream at any tick t may only depend upon values computed for previous tick $t' < t$. This is definitively not the case for LUCID (LUCID includes the inverse of \$, an ‘uncausal’ operator).
- The $8\frac{1}{2}$ stream algebra verifies the finite memory assumption: there exists a finite bound such that the number of past values that are necessary to produce the current values remains smaller than the bound.

Note that the implementation of $8\frac{1}{2}$ streams enables a static execution model: the successive values making a stream are the successive values of a single memory location and we do not have to rely on a garbage collector to free the unreachable past values (as in Haskell lazy lists, see for instance Hudak et al., 1996). In addition, we do not have to compute the value of a stream at each tick, but only at the tocks.

2.3. Combining streams and collections into fabrics

A fabric is a stream of collections or a collection of streams. In fact, we distinguish between two kinds of fabrics: static and dynamic. A static fabric is a collection of streams where every element has the same clock (the clock of a stream is the set of its tocks). In an equivalent manner, a static fabric is a stream of collections where every collection has the same geometry. Fabrics that are not static are called dynamic. The compiler is able to detect the kind of the fabric and compiles only the static ones. Programs involving dynamic fabrics are interpreted.

Collection operations and stream operations are easily extended to operate on static fabrics considering that the fabric is a collection (of streams) or a stream (of collections).

$8\frac{1}{2}$ is a declarative language: a program is a system representing a set of fabric definitions. A fabric definition takes a form similar to:

$$T = A + B \tag{1}$$

Eq. (1) is a $8\frac{1}{2}$ expression that defines the fabric T from the fabric A and B (A and B are the parameters or the inputs of T). This expression can be read as a definition (the naming of the expression $A+B$ by the identifier T) as well as a relationship, satisfied at each moment and for each collection element of T , A and B . Fig. 1 gives a three-dimensional representation of the concept of fabric.

Running a $8\frac{1}{2}$ program consists in solving fabric equations. Solving a fabric equation means ‘enumerating the values constituting the fabric’. This set of values is structured by the stream and collection aspects of the fabric: let a fabric be a stream of collections; in accordance to the time interpretation of stream, the values constituting the fabric are enumerated in the stream’s ascend-

Table 4
Example of a sampling expression

A	1	2	3	4	5	6	7	8	9
B	False	False	False	True	False	True	True	False	True
A when B				4		6	7		9

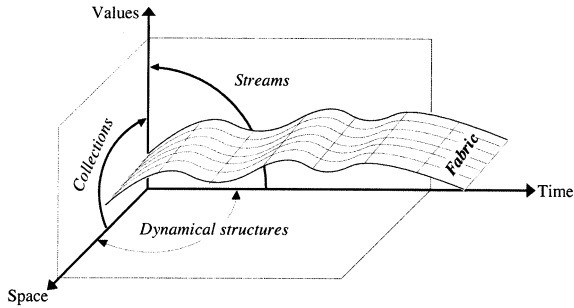


Fig. 1. A fabric specified by a $8\frac{1}{2}$ equation is an object in the (time, space, value) reference axis. A stream is a value varying in time. A collection is a value varying in space. The variation of space in time determines the dynamical structure (cf. Section 5).

ing order. So, running an $8\frac{1}{2}$ program means enumerating, in sequential order, the values of the collections making the stream. The enumeration of the collection values is not subject to some predefined order and may be done in parallel.

2.4. Recursive definitions

A definition is recursive when the identifier on the left hand side appears also directly or indirectly on the right hand side. Two kinds of recursive definitions are possible.

2.4.1. Temporal recursion

Temporal recursion allows the definition of the current value of a fabric using its past values. For example, the definition:

$$T@0 = 1 \quad T = \$T + 1 \text{ when Clock 1}$$

specifies a counter, which starts at 1 and counts at the speed of the tocks of clock 1. The @0 is a temporal guard that quantifies the first equation and means ‘for the first tock only’. In fact, T counts the tocks of Clock 1.

The order of equations in the previous program does not matter: the unquantified equation applies only when no quantified equation applies. The language for expressing guards is restricted to @n with the meaning ‘valid for the nth tock only’.

2.4.2. Spatial recursion

Spatial recursion is used to define the current value of a point using current values of other

points of the same fabric (see Fig. 2). For example:

$$iota = 0\#(1 + iota:[2]) \tag{2}$$

is a fabric with three elements such that $iota.i$ is equal to i . The operator: $[n]$ truncates a collection to n elements so we can infer from the definition that $iota$ has three elements (0 is implicitly coerced into a one-point collection). Let $\{iota_1, iota_2, iota_3\}$ be the value of the collection $iota$. The definition states that:

$$\begin{aligned} \{iota_1, iota_2, iota_3\} \\ = \{0\}\#(\{1, 1\} + \{iota_1, iota_2\}) \end{aligned}$$

which can be rewritten as:

$$\begin{cases} iota_1 = 0 \\ iota_2 = 1 + iota_1 \\ iota_3 = 1 + iota_2 \end{cases}$$

which proves our previous assertion.

We have developed the notions that are necessary to check if a recursive collection definition has a well-defined solution. The solution can always be defined as the least solution of some fixpoint equation. However, an equation like ‘ $x = \{x\}$ ’ does not define a well formed array (the number of dimensions is not finite). We insist that all elements of the array solution must be defined as in Giavitto (2000).

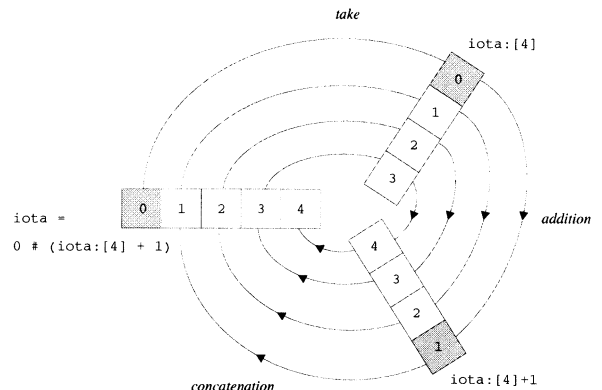


Fig. 2. Sequential computation of $iota$.

3. Examples of 8½ programs for DS with a static structure

All the examples in this section have been processed by the 8½ environment presented in Giavitto (1999) and the illustrations have been produced by the 8½–gnuplot interface.

3.1. Numerical resolution of a parabolic partial differential equation

This example is paradigmatic of a diffusion process. We want to simulate the diffusion of heat in a thin uniform rod. Both extremities of the rod are held at 0 °C. The solution of the parabolic equation:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \quad (3)$$

gives the temperature $U(x, t)$ at a distance x from one end of the rod after time t . An explicit method of solution uses finite-difference approximation of Eq. (3) on a mesh ($X_i = ih, T_j = jk$) which discretizes the space of variables, cf. Smith (1985).

One finite-difference approximation to Eq. (3) is:

$$\frac{U_{i,t+1} - U_{i,t}}{k} = \frac{U_{i+1,t} - 2U_{i,t} + U_{i-1,t}}{h^2}$$

which can be rewritten as:

$$U_{i,j+1} = rU_{i-1,j} + (1 - 2r)U_{i,j} + rU_{i+1,j} \quad (4)$$

where $r = k/h^2$. It gives a formula for the unknown temperature $U_{1,j+1}$ at the $(i, j+1)$ th mesh point in term of known temperatures along the j th time-row. Hence we can calculate the unknown pivotal values of U along the first time-row $T=k$, in terms of known boundary and initial values along $T=0$, then the unknown pivotal values along the second time-row in terms of the first calculated values, and so on (see Fig. 3 on the left).

The corresponding 8½ program is very easy to derive and describes simply the initial values, boundary conditions and the specification of the relation (Eq. (4)). The stream aspect of a fabric corresponds to the time axis while the collection aspect represents the rod discretization. The sec-

ond argument of the when operator is Clock which represents the time discretization (cf. Fig. 3). The expression 'n generates a vector of n elements where the i th element has value i .

```
start = some initial temperature distribution;
Begin = 0;
End = 0;
U@0 = start;
U = Begin#inside#End;
Float inside = 0.4*pU(left)+0.2*pU(middle)+
0.48pU(right);
pU = $U when Clock;
left = '6;
right = left + 2;
middle = left + 1.
```

3.2. The simulation of a reactive system in artificial life

Here is an example of an hybrid DS, a 'wlumf', which is a 'creature' whose behavior (eating) is triggered by the level of some internal state (see Maes, 1991 for such model in ethological simulation).

More precisely, a wlumf is hungry when its glycaemia is under 3. It can eat when there is some food in its environment. Its metabolism is such that when it eats, the glycaemia goes up to 10 and then decreases to 0 at a rate of one unit per time step. All these variables are scalar. Essentially, the wlumf is made of counters and flip–flop triggered and reseted at different rates,

```
boolean Food In Neighbourhood = Ran-
dom(bool);
```

```
System wlumf =
{
Hungry@0 = false;
Hungry = (Glycaemia < 3);
Glycaemia@0 = 6;
Glycaemia = if Eating then 10 else max (0, $
Glycaemia - 1) when Clock fi;
Eating = $Hungry && Food In Neighbor-
hood;
}
```

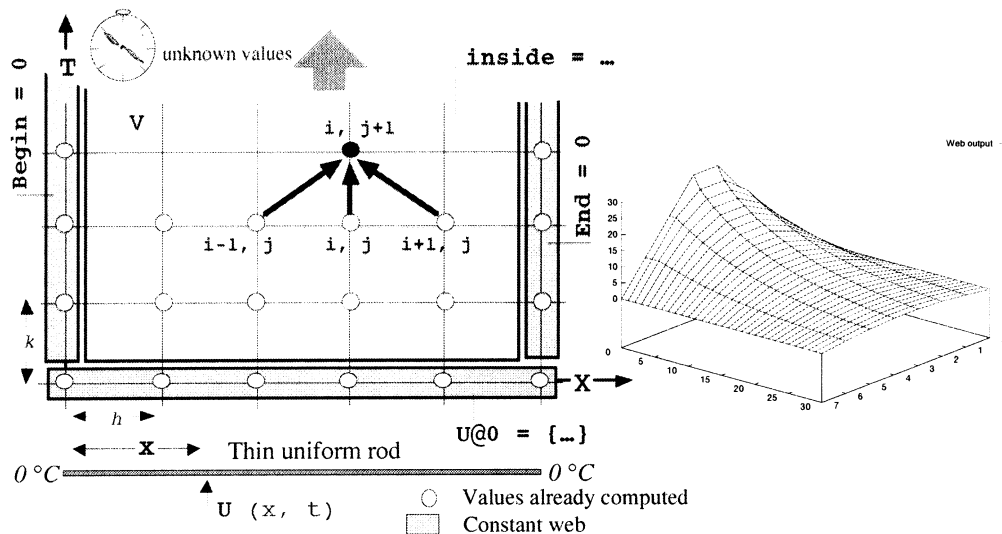



Fig. 3. Diffusion of heat in a thin uniform rod. The picture on the right is the result of the 8½ program run visualized by the 8½-gnuplot interface.

The result of an execution is given in Fig. 4.

3.3. An example of iterated equations: Turing's model of morphogenesis

A. Turing proposed a model of chemical reaction coupled with a diffusion processes in cells to explain patterns formation. The system of differential equations, from Bard and Lauder (1974), is:

$$\frac{dx_r}{dt} = \frac{1}{16}(16 - x_r y_r) + (x_{r+1} - 2x_r + x_{r-1})$$

$$\frac{dy_r}{dt} = \frac{1}{16}(16 - y_r - \beta) + (y_{r+1} - 2y_r + y_{r-1})$$

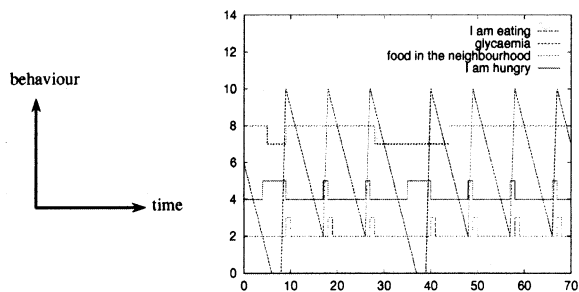


Fig. 4. Behavior of a hybrid DS.

where x and y are two chemical reactives that diffuse on a discrete torus of cells indexed by r . This model mixes a continuous phenomena (the chemical reaction in time) and a discrete diffusion process. Note that in the heat diffusion example, we consider a continuous process, which is then discretized for the purpose of numerical resolution while here the diffusion is initially in the discrete space of cells.

In 8½ we retrieve exactly the same equations dx and dy . The other equations correspond to the computation of intermediate values like $xdiff...$ to the computation of an initial value beta or the access to the neighborhood through a gather operation. Note that the corresponding c program is more than 60 lines long.

```

nbcell = 60
iota = 'nbcell; (* generates the vector {0, 1, ...,
59} *)
right = if (iota == 0) then (nbcell-1) else
(iota-1) fi
left = if (iota == (nbcell-1)) then 0 else
(iota+1) fi
rsp = 1.0/16.0
diff1 = 0.25
diff2 = 0.0625
x@0 = 4.0
    
```

$$\begin{aligned}
 x &= \$x + \$dx \text{ when Clock} \\
 y @ 0 &= 4.0 \\
 y &= \max(0.0, \$y + \$dy) \text{ when Clock} \\
 \text{beta} &= 12.0 + \text{rand}(0.05 * 2.0) - 0.05 \\
 xdiff &= x(\text{right}) + x(\text{left}) - 2.0 * x \\
 ydiff &= y(\text{right}) + y(\text{left}) - 2.0 * y \\
 dx &= \text{rsp} * (16.0 - x * y) + xdiff * diff1 \\
 dy &= \text{rsp} * (x * y - y - \text{beta}) + ydiff * diff2
 \end{aligned}$$

In Fig. 5, we have presented the results after 100 time steps (starting with a random distribution of the reactive) and after 1000 time steps when the solution has reached its equilibrium.

4. Simulation of genetic networks in 8½

Gene expression investigation by in silico methods represents one of the challenging problem of the bioinformatic. Several models have been proposed to cover different aspects of gene expression. Qualitative models appeared to encompass the main features of the regulation or decision networks. Models for gene network expressions are based on several theoretical tools: boolean networks (Thieffry and Roméro, 1999), multi-valued logic networks (Thieffry and Thomas, 1998), circuit simulation (McAdams and Shapiro, 1995), weighted matrices (Weaver et al., 1999), Petri nets (Matsuno et al., 1999), differential equations (Chen et al., 1999), etc. Genetics networks with tens to hundreds of genes are difficult to specify with currently available programming languages and require extensive programming. In addition, several hypothesis must be tested and the

resulting models have to integrate several features that do not fit into a single framework. In this context, the expressive power of the underlying simulation language is of great importance for prototyping all the variations of the models and to reduce the development time of the simulation.

In this section, we illustrate the versatility and the simplicity of the 8½ approach for the simulation of a paradigmatic example of a genetic network. For the sake of simplicity of the exposition, the models are simplifications considered in the literature of the complex interacting genetic circuit that operates in bacteriophage lambda to decide between lytic or lysogenic growth, to maintain the prophage in a lysogen, and to throw the switch during induction (a complete description is available in McAdams and Shapiro, 1995).

4.1. Boolean systems and some other discrete models

4.1.1. Boolean systems

Fig. 6 gives a simplified view of the interplay between gene *cI* and protein CRO in bacteriophage lambda. As usual, an arrow \rightarrow represents a positive feedback while a stopped link \dashv represents a negative one.

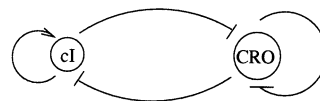


Fig. 6. Very simplified form of interaction between gene *cI* and protein CRO.

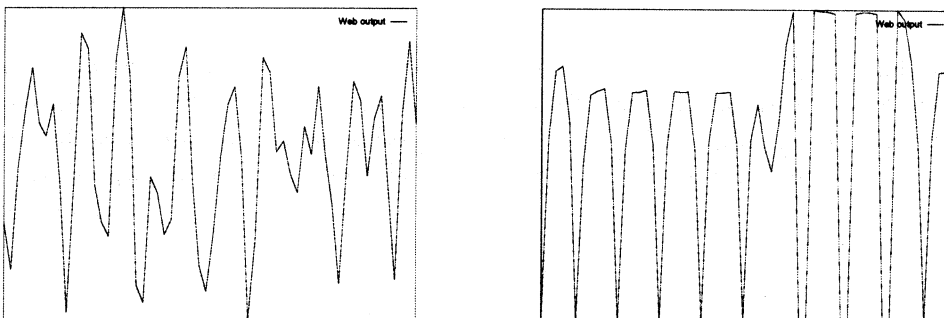


Fig. 5. Diffusion/reaction in a Torus.

A first approach models the expression of the genes and the level of the protein by a boolean. Table 5 gives an evolution of cI and CRO which is compatible with the qualitative constraints given by Fig. 6. Variable X represents the boolean value associated with product X and $\$X$ represents the value of product X at the previous time step. Two transitions for CRO are compatible with the given constraint when the system is in state $(cI, CRO) = (1, 1)$. The two possibilities are labeled α and β .

The corresponding $8\frac{1}{2}$ program is straightforward.

$CRO@0 = \dots$ (* some initial value *); CRO, not ($\$CRO$) when Clock 1; $cI@0, \dots; cI_\alpha = \cI or not ($\$CRO$); $cI_\beta = \$cI$.

Note that CRO is stated equal to “not $\$CRO$ ” and then can be used in place of this expression (this property is termed as ‘transparential referency’ in dataflow languages). In consequence, the equation for cI_α can be rewritten $cI_\alpha = CRO$ which shows that the expression of cI depends ‘instantaneously’ from the level of CRO. The rate of change is fixed by Clock 1 and imposed to CRO using a trigger operator. This clock is also the clock for cI because of the dependency between cI and CRO. However, it is very easy to give another rate of evolution by using a trigger in the rule for cI . By using different clocks, we can easily model different rates of change.

4.1.2. Discrete state systems

The previous model is too rough: we cannot express that CRO represses itself only when its level is high enough. We have to adopt a finer representation for the levels of CRO see Thieffry and Thomas (1998). Table 6 gives a possible transition table for cI and CRO when cI is represented as a boolean and CRO takes a level

Table 6

A possible transition table for cI and CRO when $cI \in \{0, 1\}$ and $CRO \in \{0, 1, 2\}$ and the associated $8\frac{1}{2}$ program

$\$cI$	$\$CRO$	cI	CRO
0	0	1	2
0	1	0	2
0	2	0	0 or 1
1	0	1	0
1	1	0	0
1	2	0	0

value in $\{0, 1, 2\}$. Here too, the corresponding $8\frac{1}{2}$ is straightforward. If the coding of a transition table into a function is a burden, the transition table can be specified as such by a $8\frac{1}{2}$ collection, and a selection operator is simply used to compute the transition by looking in this table.

$CRO@0, \dots; cI@0, \dots; cI$, if ($\$CRO = 0$) then 1 else 0 fi; when Clock 1; CRO = if $\$cI = 1$ then 0; else if ($\$CRO \leq 1$) then 2 else 0 or 1 fi fi; when Clock 1.

4.1.3. Asynchronous systems diffusion, continuous models, etc

It is very unlucky that two products change their state synchronously. We then have to render the fact that only one variable changes its state at a time. We suppose that the probability for CRO to change its state is p_{CRO} and probability for cI is p_{cI} (it is not mandatory that $p_{CRO} + p_{cI} = 1$).

Asynchronous iteration are also handled very simply in $8\frac{1}{2}$ because it is possible to produce a clock with a probabilistic tock rate of p with the ‘Rclock p ’ construct. To take into account the asynchronous change, just replace the clocks appearing in the previous program by ‘Rclock p_{cI} ’ and accordingly.

Another problem is to take into account the diffusion of the products. For instance, there is a delay between the beginning of the production of cI and the repression for CRO by cI . This can be modeled using additional delay operator ‘ $\$$ ’. One

Table 5

Possible evolution functions for cI and CRO

		$\$CRO$				$\$CRO$	
		CRO				cI	
		0	1			0	1
$\$cI$	0	1	0	$\$cI$	0	1	0
	1	1	0		1	1	$0_\alpha/1_\beta$

Value 0 and 1 represents boolean false and true, respectively

‘\$’ refers to the previous time step, two ‘\$’ refer to the event (or time step) preceding the previous one, etc.

Numerous others formalisms have been used for genetic networks, ranging from Petri Nets, e.g. Hofestädt (1994) and Matsuno et al. (1999) to hybrid systems mixing differential equations and boolean states, e.g. McAdams and Shapiro (1995). We have already show in the previous section the ability of \mathcal{S} to express this last kind of model. In particular, we are confident that a language like \mathcal{S} is very well-fitted to express the circuit diagrams used in McAdams and Shapiro (1995) because declarative language have already been successful in the domain of electric circuit simulation.

5. Examples of dynamical systems with a dynamical structure

Fabrics with a static structure cannot describe phenomena that grow in space, like plants. To describe those structures, we need dynamically structured fabrics. The rest of this section gives some examples of the kind of dynamics fabrics we can achieve in \mathcal{S} . Note that we do not need to introduce new operators, the current definitions of fabrics already enable the construction of dynamically shaped fabrics. However, some examples are not easily stated in the current \mathcal{S} version. This will be discussed in the last section.

5.1. Pascal’s triangle

This somewhat artificial example is a pretext to introduce growing collections. The numbers in Pascal’s triangle give the binomial coefficients. The value of the point (row, col) in the triangle is the sum of the values of the point (row−1, col) and the point (row−1, col−1). We decide to map the rows in time, thus the fabric representation of Pascal’s triangle is a stream of growing collections. This fabric is dynamic because the number of elements in the collection varies in time.

We can identify that the row l ($l > 0$) is the sum of row $(l-1)$ concatenated with 0 and 0 concatenated with row $(l-1)$. The \mathcal{S} program is

straightforward:

```
t = ($t#0) + (0#$t) when Clock;
t@0 = 1
```

The five first values of Pascal’s triangle are:

```
Tock: 0:{1}: int[1]
Tock: 1:{1, 1}: int [2]
Tock: 2:{1, 2, 1}: int[3]
Tock: 3: {1, 3, 3, 1}: int[4]
Tock: 4: {1, 4, 6, 4, 1}: int[5]
```

5.2. Eratosthenes’s sieve

We present a modified version of the famous Eratosthenes’s sieve to compute prime numbers. This example is adapted from a paradigmatic example in artificial chemistry, cf. Dittrich (2001) (originally it relies on a multiset of numbers and we use here a vector of numbers).

The Eratosthenes’s sieve consists of a generator producing increasing integers and a list of known primes numbers (starting with the single element 2). Each time we generate a new number, we try to divide it by all currently known prime numbers. A number that is not divided by a prime number is a prime number itself and is added to the list of prime numbers.

Generator is a fabric that produces a new integer at each tock. Extend is the number generated with the same size as the fabric of already known prime numbers. Modulo is the fabric where each element is the modulo of the produced number and the prime number in the same column. Zero is the fabric containing boolean values that are true every time that the number generated is divided by a prime number. Finally, reduced is a reduction with an or operation, that is, the result is true if one of the prime numbers divides the generated number. The $x:|y|$ operator shrinks the fabric x to the rank specified by y . The rank of a collection is a vector where the i th element represents the number of elements of x in the i th dimension.

```
Generator@0 = 2;
Generator = $ generator + 1 when Clock;
Extend = generator: |Scribe|;
```

```
Modulo = extend%$Scrible;
Zero = (modulo = = (0: |modulo|));
Reduced = or\zero;
crible = $Scrible#generator when (not reduced);
crible@0 = generator;
```

The five first steps of the execution give for crible:

```
Tock: 0:{2}: int[1]
Tock: 1:{2, 3}: int[2]
Tock: 2:{2, 3}: int[2]
Tock: 4:{2, 3, 5}: int[3]
```

5.3. Coding D0L-systems

An *L* system is a parallel rewriting system (every production rule that might be used at each derivation state are used simultaneously) developed by Lindenmayer in the 1960s, cf. Lindenmayer (1968). It has since become a formalism used in a wide range of applications from the description of cellular interactions to a model of parallel computation, e.g. Prusinkiewicz and Hannan (1992).

The parallel derivation process used in the *L* systems is useful to describe processes evolving simultaneously in time and space (growth processes, descriptions and codings of plants and plants development, etc.). To describe a wide range of phenomena, *L* systems of many different types have been designed. We will restrict ourselves to the simplest form of *L* systems: D0L systems.

Formally, a D0L system is a triple $G = (\Sigma, h, \omega)$ where Σ is an alphabet, h is a finite substitution on Σ (into the set of subsets of Σ^*) and ω , referred to as the axiom, is an element of Σ^+ .

The *D* letter stands for deterministic, which means there exists at most a single production rule for each element of Σ . Therefore, the derivation sequence is unique while in nondeterministic *L* systems (since there can be more than one production rule applied at each derivation state), there exists more than one derivation sequence. The numerical argument of the *L* system gives the number of interactions in the rewriting process; therefore, a 0*L* system is a context free *L* system (whereas an *nL* system is context sensitive with *n* interactions).

An example of *L* system: the development of a one-dimensional organism. We consider the development states of a one-dimensional organism (a filamentous organism). It will be described through the definition of a 0*L* system. Each derivation step will represent a state of development of the organism. The production rules allow each cell to remain in the same state, to change its state, to divide into several cells or to disappear.

Consider an organism where each cell can be in one of two states *a* and *b*. The *a* state consists of dividing itself whereas the *b* state is a waiting state of one division step.

The production rules and the five first derivation steps are:

```
 $\omega: b_r; t_0: b_r$ 
 $p_1: a_r \rightarrow a_l b_r; t_1: a_r$ 
 $p_2: a_l \rightarrow b_l a_r; t_2: a_l b_r$ 
 $p_3: b_r \rightarrow a_r; t_3: b_l a_r a_r$ 
 $p_4: b_l \rightarrow a_l; t_4: a_l a_l b_r a_l b_r$ 
```

The cell polarity, which is a part of the cell state is given with the *l* and *r* indice. A derivation tree of the process is detailed in the Fig. 7 (partly taken from Lindenmayer and Jürgensen, 1992). The polarity changing rules of this example are very close to those found in the blue–green bacterium *Anabaena catenula* described in Mitchinson and Wilcox (1972) and Koster and Lindenmayer (1987). Nevertheless, the timing of the cell division is not the same.

The implementation of the production rules in 8½ is straightforward. Through a direct transla-

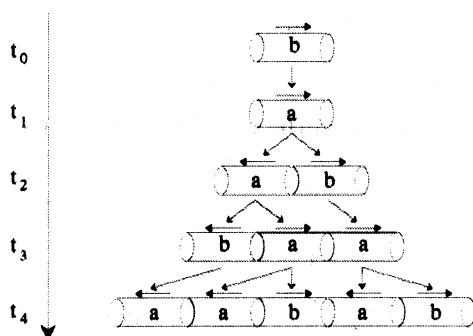


Fig. 7. The first derivations of the *A. catenula* (the cell polarity is indicated with an upper arrow).

tion of the rules, we have the following 8½ program:

```
w = a_r;
a_r = $a_l # $b_r when Clock; a_r@0 = {'a_r'};
a_l = $b_l # $a_r when Clock; a_l@0 = {'a_l'};
b_r = $a_r when Clock; b_r@0 = {'b_r'};
b_l = $a_l when Clock; b_l@0 = {'b_l'};
```

The five first steps of the execution are:

```
Tock: 0: {b_r}: char[1]
Tock: 1: {a_r}: char[1]
Tock: 2: {a_l, b_r}: char[2]
Tock: 3: {b_l, a_r, a_r}: char[3]
Tock: 4: {a_l, a_l, b_r, a_l, b_r}: char[5]
```

More generally, it is possible to describe the whole class of D0L systems in 8½ even the non propagating D0L systems, see Michel (1996).

6. Conclusions and perspectives

The 8½ is a long term effort to validate the effectiveness of declarative language in the simulation of DS. The original motivation was the simulation of some large DSs found in physics. We can summarize the lessons of the 8½ project by the following points:

- 1) The declarative style is effective in providing a framework close to the usual (mathematical) models used by an end-user, if the data and constructs offered by the language correspond to the ground concepts used in the application domain.
- 2) Smart interpreters and compilers are good! They relieve the programmer from many burden and ensure many consistency checks. For instance, in 8½ several non-standard type-inference systems are used to derive the shape of the specified collections, to use a static and more efficient simulation algorithm or a dynamic one, and to check causality between the variables in the equations.
- 3) The declarative language does not imply an unacceptable loss of efficiency. For instance, we have developed some compilation techni-

ques that reduce the loss of efficiency to 30% in the example of the heat diffusion compared with a hand-coded C program.

- 4) The declarative style does not constrain the parallelization. For instance, 8½ collections are well fitted for the expression of the data-parallelism, see De Vito and Michel (1996) and Giavitto et al. (1998). More generally, declarative languages are well-fitted for the minimal expression of sequencing in a program, leading to a maximal amount of (implicit) parallelism. However, the exploitation of this parallelism can be as hard as in the imperative case.

It is only recently that DSs model of biological processes have drawn our attention. The examples sketched in this paper show that the 8½ approach can be relevant for this kind of systems too. However, except in Section 5, all the examples used exhibit a static structure.

By a static structure, we mean that the phase space of the DS can be known statically before the simulation. It is precisely the shape-inference phase of a 8½ program that determines this phase space. The case of the examples in Section 5 is more difficult: a precise phase space cannot be inferred before the simulation run, but the general form with some parameters is known at compile-time and the parameters are derived at run-time (e.g. when it is sufficient to work with unbounded lists instead of fixed-size vectors).

There is, however, a kind of DS that is very uneasy to model in 8½: systems that have an intrinsic dynamical structure. Examples of this kind are *P* systems with active membrane described in Paun (2000) or multi-agent systems with dynamic creations and mobility. The restriction of collections to the array structure is also problematic and there is an urgent need for more sophisticated aggregation structures. To face these problems, new concepts have to be introduced. Some extensions have been proposed in Michel and Giavitto (1998a), but the result is too much targeted to a family of application (those whose topology is built as a bottom-up tree). This motivates the beginning of a new project consider-

ing more sophisticated topology and dynamic constructs.

Acknowledgements

The authors would like to thank J. Cohen and the members of the “Simulation and Epigenesis” group at GENOPOLE-Evry for fruitful discussions, biological motivations and challenging questions. The friendly atmosphere of the IPCAT workshop has also raised many questions that have suggested many developments and rethinking. This research is supported in part by the CNRS, the GDR ALP, IMPG and GENOPOLE/University of Evry.

References

- Arvind, Brock, J.D., 1983. Streams and managers. In: Proceedings of the 14th IBM Computer Science Symposium.
- Bard, J., Lauder, L., 1974. How well does Turing’s theory of morphogenesis work. *Journal of Theoretical Biology* 45, 501–531.
- Blelloch, G., 1989. Scans as primitive parallel operations. *IEEE Transactions on Computers* 38 (11), 1526–1538.
- Blelloch, G., Sabot, G.W., 1990. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing* 8, 119–134.
- Chen, T., He, H., Church, G., 1999. Modeling gene expression with differential equations. In: Proceedings of the Pacific Symposium on Biocomputing’99, pp. 112–123.
- De Vito, D., Michel, O., 1996. Effective SIMD code generation for the high-level declarative data-parallel language δ_2^1 . In: EuroMicro’96. IEEE Computer Society, pp. 114–119.
- Dittrich, P., 2001. Artificial chemistry webpage. URL <http://www.cs.uni-dortmund.de/achem>.
- Fontana, W., Buss, L., 1996. The barrier of objects: from dynamical systems to bounded organizations. In: Casti, J., Karlqvist, A. (Eds.), *Boundaries and Barriers*. Addison-Wesley, pp. 56–116.
- Giavitto, J.-L., 1992. Typing geometries of homogeneous collection. In: Second International Workshop on Array Manipulation (ATABLE), Montréal.
- Giavitto, J.-L., 1999. Scientific report for the HDR. Ph.D. thesis. LRI, Université de Paris-Sud, Centre d’Orsay, Research Report 1226.
- Giavitto, J.-L., 2000. A framework for the recursive definition of data structures. In: ACM-Sigplan Second International Conference on Principles and Practice of Declarative Programming (PPDP’00). ACM-Press, Montréal, pp. 45–55.
- Giavitto, J.-L., De Vito, D., Sansonnet, J.-P., Sep. 1998. 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.
- Giavitto, J.-L., Michel, O., Sansonnet, J.-P., 1995. Group based fields. In: Takayasu, L., Halstead, R.H.J., Queinnec, C. (Eds.), *Parallel Symbolic Languages and Systems (International Workshop PSLs’95)*, Lecture Notes in Computer Science, vol. 1068. Springer, Beaune, France, pp. 209–215.
- Hofestädt, R., 1994. A Petri net application of metabolic processes. *Journal of System Analysis Modelling and Simulation* 16, 113–122.
- Hudak, P., et al., 1996. Report on the programming language HASKELL a non-strict, purely functional language, version 1.3. CS Department, Yale University.
- Jayaraman, B., 1992. Implementation of subset-equational program. *Journal of Logic Programming* 12, 299–324.
- Koster, C.G., Lindenmayer, A., 1987. Discrete and continuous models for heterocyst differentiation in growing filaments of blue-green bacteria. *Acta Biotheoretica* 36, 249–273.
- Lindenmayer, A., 1968. Mathematical models for cellular interactions in development parts I and II. *Journal of Theoretical Biology* 18, 280–315.
- Lindenmayer, A., Jürgensen, H., 1992. Grammars of development; discrete-state models for growth, differentiation, and gene expression in modular organisms. In: Ronzenberg, G., Salomaa, A. (Eds.), *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*. Springer, pp. 3–21.
- Lisper, B., 1993. On the relation between functional and data-parallel programming languages. In: Proceedings of the Sixth International Conference on Functional Languages and Computer Architectures, ACM, ACM Press.
- Maes, P., 1991. A bottom-up mechanism for behavior selection in an artificial creature. In: Book, B. (Ed.), *Proceedings of the First International Conference on Simulation of Adaptive Behavior*. MIT Press.
- Mahiou, A., Giavitto, J.-L., 1994. Data-parallelism and data-flow: automatic mapping and scheduling for implicit parallelism. In: Franco-British meeting on Data-parallel Languages and Compilers for portable parallel computing, Villeneuve d’Ascq, 20 avril.
- McAdams, H., Shapiro, L., 1995. Circuit simulation of genetic networks. *Science*, 269.
- Michel, O., 1996. A straightforward translation of D0L Systems in the declarative data-parallel language δ_2^1 . In: Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y. (Eds.), EuroPar’96 Parallel Processing, Lecture Notes in Computer Science, vol. 1123. Springer, pp. 714–718.
- Michel, O., Giavitto, J., 1994. Design and implementation of a declarative data-parallel language. In: Post-ICLP’94 Workshop W6 on Parallel and Data Parallel Execution of Logic Programs. Uppsala University, Computing Science Department, S. Margherita Liguria, Italy.
- Michel, O., Giavitto, J.-L., 1998a. Amalgams: Names and name capture in a declarative framework. Tech. Rep. 32, LaMI-

- Université d'Évry Val d'Essonne, also available as LRI Research-Report RR-1159.
- Michel, O., Giavitto, J.-L., 1998b. A declarative data parallel programming language for simulations. In: Proceedings of the Seventh International Colloquium on Numerical Analysis and Computer Science with Applications. Plovdiv, Bulgaria.
- Michel, O., Giavitto, J.-L., Sansonnet, J.-P., 1994. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In: SMS-TPE'94: Software for Multiprocessors and Supercomputers, Office of Naval Research USA & Russian Basic Research Foundation, Moscow, 21–23 September, pp. 103–111.
- Matsuno, H., Doi, A., Nagasaki, M., Miyano, S., 1999. Hybrid Petri Net representation of gene regulatory network. In: Proceedings of the Pacific Symposium on Biocomputing'99, pp. 112–123.
- Mitchinson, G.J., Wilcox, M., 1972. Rule governing cell division in *anaeba*. *Nature* 239, 110–111.
- NSF, 1991. Grand challenges: high performance computing and communications. A Report by the Committee on Physical, Mathematical and Engineering Sciences, NSF/CISE, 1800 G Street NW, Washington, DC 20550.
- Paun, G., 2000. From cells to computers: computing with membranes (p systems). In: Workshop on Grammar Systems. Bad Ischl, Austria.
- Prusinkiewicz, P., Hanan, J., 1992. L systems: from formalism to programming languages. In: Ronzenberg, G., Salomaa, A. (Eds.), *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*. Springer, pp. 193–211.
- Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E., 1986. Programming with sets: and introduction to SETL. Springer.
- Sipelstein, J.M., Belloch, G., 1991. Collection-oriented languages. *Proceedings of the IEEE* 79 (4), 504–523.
- Smith, G.D., 1985. Numerical solution of partial differential equations: finite difference methods. In: *Oxford Applied Mathematics and Computing series*. Oxford University Press.
- Thieffry, D., Thomas, R., 1998. Qualitative analysis of gene networks. In: Proceedings of the Pacific Symposium on Biocomputing'98, pp. 77–88.
- Thieffry, D., Roméro, D., 1999. The modularity of biological regulatory networks. *Biosystem* 50, 49–59.
- TMC, 1986. *The Essential *LISP Manual*. Thinking Machines Corporation, Cambridge, MA.
- Tofooli, T.N.M., 1987. *Cellular Automata Machine*. MIT Press, Cambridge, MA.
- Wadge, W.W., Ashcroft, E.A., 1976. LUCID—a formal system for writing and proving programs. *SIAM Journal on Computing* 3, 336–354.
- Weaver, D., Workman, C., Stormo, G., 1999. Modeling regulatory networks with weight matrices. In: Proceedings of the Pacific Symposium on Biocomputing'99, pp. 112–123.