Abstract—Modern programming languages allow the definition
and the use of arbitrary nested data structures but this is not
generally considered in unconventional programming models. In
this paper, we present arbitrary nesting in MGS, a spatial comput-
ing language. By considering different classes of neighborhood
relationships, MGS can emulate several unconventional computing
models from a programming point of view. The use of arbitrary
nested spatial structures allows a hierarchical form of coupling
between them. We propose an extension of the MGS pattern-
matching facilities to handle directly nesting. This makes possible
the straightforward emulation of a larger class of unconventional
programming models.

I. INTRODUCTION

Modern programming languages allow data structure to be
nested so that a valid element of a structure can also be in
its turn another structure. Generally, this is not considered in
unconventional programming models. For instance, the state of
a cell in a cellular automata is not (the state of) another cellular
automata. Another example: the value labeling a symbol in
parametric Lindenmayer system is not (a string representing a
derivation in) another Lindenmayer system.

In chemical computing, as exemplified in Gamma [1],
chemical solutions are abstracted as multisets (a generalization
of the notion of set in which members are allowed to appear
more than once) and a molecule corresponds to an elementary
data and not another chemical solution. Nested multisets
are considered in membrane systems\(^1\) [4] but are studied
as a completely different computational model. Indeed, the
management of the nesting entails the introduction of new
mechanisms (transport rules in the case of membrane systems).

In this paper we consider arbitrary nesting in MGS, a spatial
computing language where space is managed through the
structure of the data. MGS relies on neighborhood relationships
to represent physical (spatial distribution, localization of the
resources) or logical constraints (inherent to the problem to be
solved) in a computation.

By considering different classes of neighborhood relation-
ships, MGS can emulate several unconventional computing
models from the point of view of the programming. The
use of arbitrary nested spatial structures allows a hierarchical
form of coupling between them. Furthermore, we propose
an extension of the MGS pattern-matching facilities to handle
nesting explicitly. This makes possible the concise expression
of various algorithms as well as the straightforward emulation
of a larger class of unconventional programming models.

Outlines: This paper is organized as follows. The next
section introduces the emerging field of spatial computing
and the notions of topological collection and transformation
developed in MGS. Section III discusses the relevance of nesting in
the context of spatial computing and section III-D proposes
a new pattern matching construct to make the handling of
nested structures mores easier. Section IV exemplifies the
use of nested spaces with three direct applications. The first
encodes terms used to represents boolean formulae with nested
sets. The computation of a disjunctive normal form on this
representation is explained. The second example computes
quadrees, a recursive data structure for partitioning a two
dimensional space. The last one is dedicated to the informal
translation of the fraglet computational model into MGS.
Related and future work concludes this article.

II. BACKGROUND

A. Computing in Space, Space in Computation and Spatial
Computing

Spatial Computing is an emerging field of research [5]
where the computation is structured in term of spatial rela-
tionships where only “neighbor” elements may interact.

For example, the elements of a physical computing system
are spatially localized and when a locality property holds, only
elements that are neighbor in space can interact directly. So
the interactions between parts are structured by the spatial
relationships of the parts.

Even for non physical system, usually an element does not
interact with all other elements in the system. For instance,
from a given element in a data structure, only a limited number
of other elements can be accessed [6]: in a simply linked list,
the elements are accessed linearly (the second after the first,
the third after the second, etc.); from a node in a tree, we can access the father or the sons; in arrays, the accessibility relationships are left implicit and implemented through incrementing or decrementing indices (called “Von Neumann” or “Moore” neighborhoods if one or several changes are allowed).

More generally, if an element \( e \) in a system interacts during a computation with a subset \( E = \{e_1, \ldots, e_n\} \) of other elements, it also interacts with any subset \( E' \) included in \( E \). This closure property induces a topological organization: the set of elements can be organized as an abstract cellular complex which is a spatial representation of the interactions in the computation [7]. This abstract space instantiates a neighborhood relationship that represents physical (spatial distribution, localization of the resources) or logical (inherent to the problem to be solved) constraints.

In addition, space can be an input to computation or a key part of the desired result of the computation, e.g. in computational geometry applications, amorphous computing [8], claytronics [9], distributed robotics or programmable matter... to cite a few examples where notions like position and shape are at the core of the application domain.

B. The MGS approach to Spatial Computing

The MGS project recognizes that space is not an issue to abstract away but that computation is performed distributed across space and that space, either physical or logical, serves as a mean, a resource, an input and an output of a computation.

1) Topological Collections: In MGS, the notion of space is handled through a slight generalization of the notion of field. In physics, a field assigns a quantity to each point of a spatial domain [10].

MGS handles spatial domains defined by abstract cellular complex [11]. An abstract cellular complex is a formal construction that builds a space in a combinatorial way through more simple objects called topological cells. Each topological cell abstractly represents a part of the whole space: points are cells with dimension 0, lines are cells with dimension 1, surfaces are 2 dimensional cells, etc. The structure of the whole space, corresponding to the partition into topological cells, is considered through incidence relationships, relating a cell and the cells in its boundary.

In this approach a field is a finite labeling of a cellular complex: a cellular complex may count an infinite number of cells but MGS restricts itself on fields labeling only a finite number of these cells. Such fields are called topological collections to stress the importance of the neighborhood relationships induced by the incidence relationships. Topological collections are a weakening of the notion of topological chain developed in algebraic topology [12] and have been introduced in [13] to describe arbitrary complex spatial structures that appear in biological systems [14] and other dynamical systems with a time varying structure [15], [16]. Topological collections generalize fields because they associate a quantity with 0-cells (points in space) but also with arbitrary \( n \)-cells.

Graphs are examples of one dimensional cellular complexes: they are made of only 0- and 1-cells. In this paper, we will stick to topological collections where the underlying complex is a graph. In [6] it has been showed how usual data structures (sets, multisets, lists, trees, arrays...) can be seen as one dimensional topological collections: the elements in a data structure are the quantities assigned by the field to the nodes of a graph.

A specific neighborhood relationship has also a special importance in the rest of this paper: the full relation. With this relation, every node is neighbor of every other nodes. This corresponds to a node-labeled complete graph and to the multiset data structure.

2) Transformations: Usually in Physics, fields and their evolution are specified using differential operators. MGS generalizes these operators in a rewriting mechanism, called transformation. A transformation is the application of some local rules following some strategy. The application of a local rule \( a \rightarrow b \) in a collection \( C \):

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

A local rule specifies a local evolution of the field: the left hand side (lhs) of the rule typically matches elements in interaction and the right hand side (rhs) computes local updates of the field.

Patterns \( \text{pat} \) are expressions defined inductively starting from the pattern variables “\( x \)” (matching exactly one element in a collection) and several operators used to compose more complex patterns: the neighborhood between two subcollections “\( \text{pat}_1, \text{pat}_2 \)” , the repetition “\( \text{pat}^* \)” , the guard “\( \text{pat}/\text{exp} \)” and the typing “\( \text{pat}:T \)” (elements matched by \( \text{pat} \) have to be of type \( T \)).

Transformations are a powerful means to define functions on topological collections complying with the underlying spatial structure. For instance, discrete analog of differential operators can be defined using transformations [17]. For multisets, transformations reduce simply to associative-commutative rewriting [18] also called multiset rewriting.

III. Nested Spaces

In classical Physics, a field can be classified as a scalar field or a vector field according to whether the value of the field at each point is a scalar or a vector. However, it is not usual to consider “field valued fields”. From the MGS perspective, this notion simply corresponds to the idea of nested topological collections. Such a feature is relevant and valuable in at least three areas: for the representation of and the computation on hierarchical or inductive data structures; in the modeling and the simulation of multiscale systems; and in the emulation of “stratified” computational models.

A. Inductive Data Structures

The possibility to nest arbitrarily data structures is now pervasive in modern programming languages. Its usefulness to represent hierarchical data (e.g., XML) or inductive structure
(e.g., list, trees) is well established. We give an example of the use of nested spaces in an algorithmic application relying on multisets in section IV-A.

B. Multiscale Systems

The modeling of a natural system often implies entities appearing on distinct temporal and spatial scales: each level addresses a phenomenon over a specific window of length and time. These scales appear for logical reasons (at a particular scale, the system exhibits uniform properties and can be modeled by homogeneous rules acting on objects relevant at this scale) or for efficiency reasons (e.g., the reductionist simulation of the whole system from first principles is computationally not tractable while we are only interested in coarse-grained description).

Multiscale models and simulations arise when interactions between scales must be considered. For spatial scales, it means that simultaneous spatial representations must be managed as in adaptive mesh refinement [19]. This method relies on a sequence of “nested rectangular grids” on which a PDE is discretized. It is important to realize that these subgrids are not patched into the coarse grid but overlaid to track the feature discretized. It is important to realize that these subgrids are not patched into the coarse grid but overlaid to track the feature of interest. A simplistic example is presented in section IV-B. Another example, in the area of discrete modeling, is the complex automata framework [20] corresponding to a “graph of cellular automata”.

Sometimes scales can be separated, meaning that the coupling between scales can be localized at some isolated interaction points in space and time. Then, the resulting computation corresponds to a hierarchical process with a directed flow of information. This is not always the case and we will introduce a dedicated pattern-matching mechanism in section III-D to ease the reference between scales.

C. Stratified Computational Models

Some models of computation exhibit naturally an inductive structure. For instance, the state of a membrane system is a multiset of symbols and (inductively) membrane systems. This structure leads directly to a nested organization of “multiset of symbols and multisets”.

Some computational models are also best described as a combination of two paradigms: the second being substituted for some generic parts in the first. We list a few examples issued from various compartmentalization devices introduced over a basic chemical framework. In membrane systems, strings have been considered instead of symbols [21]. This leads obviously to “multiset of sequences of symbols and multisets”. Nested multisets are restricted to the description of membranes organized by inclusion only. Tissue P systems [22] arrange the membranes and their interactions following an arbitrary graph, calling for a “graph of multisets”. Spatial P systems [23] are a variant of P systems which embodies the concept of space and position inside a membrane. Membranes and objects are positioned in a two-dimensional discrete space. Hence, we have to consider “grids of multisets and grids and symbols”.

In section IV-C, we will sketch the encoding of fraglets, a molecular biology inspired execution model for computer communications leading to “graph of multisets of sequences”.

D. Matching Nested Structures

In the next section, we give examples of the three usages of nested spaces we have identified above section III. The use of nested spaces does not require a priori new control structures. For example, if the reactions between symbols of a P systems are coded by a transformation EvalRule, then we can define a function Apply and an auxiliary transformation ApplyNested to thread EvalRule over the nested structure:

\[
\text{fun Apply}(x) = \text{EvalRule}(\text{ApplyNested}(x))
\]

\[
\text{and trans ApplyNested} = \text{bag} \Rightarrow \text{Apply}(x)
\]

This piece of code is enough to trigger the chemical rules specified by EvalRule through the entire structure. But the transport rules, used for example to expel one molecule from a membrane to the enclosing one, are a little bit heavier to write because they imply the simultaneous matching of two levels in the nested structure.

The usual MGS pattern constructions are “flat”: the pattern variables of a transformation \( T \) refer to elements of the collection on which \( T \) is applied [24]. To make easier the handling of nested collections, we extend the current syntax with a new construction allowing references to elements of a nested collection. The pattern construct

\[
[ \text{pat} | x ]
\]

matches a collection \( C \) nested within the current one. The pattern \( \text{pat} \) must match a subcollection \( C' \) in \( C \) and the variable \( x \) is bound to the collection \( C \) deprived of \( C' \). For example, the pattern

\[
x, [2, 3 | y] \text{ as } z
\]

matches only one occurrence in the sequence

\[
(0, (1, 2, 3, 4), 5)
\]

and binds \( x \) to 0, \( z \) to the nested sequence \((1, 2, 3, 4)\) and \( y \) to the sequence \((1, 4)\). The notation \([\text{pat} | \ldots]\) can be used to spare a variable if the rest of the subcollection is not used elsewhere.

IV. COMPUTING WITH NESTED COLLECTIONS

A. Disjunctive Normal Form

Since the logical conjunction and disjunction operators are associative, commutative and idempotent, a logical formula can be encoded by nested sets. Let consider the following type declaration:

\[
\text{type formula} = \text{string} \mid \text{Not} \mid \text{And} \mid \text{Or}
\]

\[
\text{and record Not} = \{ f: \text{formula} \}
\]

\[
\text{and collection And} = \text{set}[\text{formula}]
\]

\[
\text{and collection Or} = \text{set}[\text{formula}]
\]

In this declaration, formula is a sum type: a formula is either a boolean variable (represented by a string value), or the negation of a formula nested in a record with one field \( f \), or the conjunction (resp. disjunction) of formulae nested in a set with subtype And (resp. Or). With these types, the formula \( \neg(p \land q) \lor r \) can be represented as follows:
ogy is specified through a group presentation. Here, the GBF
GBF are collections with a regular neighborhood whose topol-
definitions:

aligned along a curve.

points in each region (we take

subdivision is recursively iterated until there is less than

n

subdivide a rectangular spatial domain into four regions. The

partitions a set of points in 2D space. Quadtrees recursively

joins are used to append two collections and

results;

inserts an element

s

are the usual map

of a well

transformation:

Grid = C

v

is associated with the cell

i

where the value

c

is

and

starting from

n

in the following).

The recursive subdivision is computed by the following transformation:

Grid specifies a 2 × 2 torus with two directions n and e. Such

grid describe a partition in four adjacent regions.

The recursive subdivision is computed by the following transformation:

trans MakeQuadTree =

c:Cloud / size(c) > 2 ⇒

MakeQuadTree (SplitCloud (c))

The function SplitCloud makes the real work:

fun SplitCloud (c:Cloud) =

let g = barycenter (c) in

let c0, c1 = split (λp.p.x<g.x, c) in

let c00, c01 = split (λp.p.y<g.y, c0) in

let c10, c11 = split (λp.p.y<g.y, c1) in

Grid: (c00@c0, c01@c e, c10@c n, c11@c (n+e))

Function SplitCloud divides a cloud of points c into four sub-

clouds cij, depending on the positions of the points compared to

the barycenter (above or below, on the left or on the right). Then

it builds a new Grid collection where the four cells are

labeled by the clouds cij. Function barycenter computes

the center of mass of a cloud of points (it is easily expressed

using a fold over the elements of the set.) The function split

takes a predicate and a collection, and returns two collections:

the elements of the first one satisfy the predicate and the

second one gathers the remaining elements. The syntactic

construction T:(...νi@cij..) builds a collection of type T

where the value νi is associated with the cell cij.

The process is illustrated on figure 1. There is no need of

the new pattern matching construct because there is no need to

“mix” the elements of a top level collection with the elements

of a nested one. It is the recursive calls of MakeQuadTree

that create the nested structure from a flat cloud of points.

\[ f = "p"::"q"::And:(() ::"r"::Or:()\]

denotes the empty collection of type C.

The computation of the disjunctive normal form can be

achieved by iterating until a fixpoint is reached, the following

trans \(DNF = \{\)

(* Simplifying unaries *) \[
\begin{align*}
&[x|\ldots]:Not|\ldots]:Not \implies x \\
&x:\text{And} / size(x)=1 \implies \text{choose}(x) \\
&x:\text{Or} / size(x)=1 \implies \text{choose}(x)
\end{align*}
\]

(* Flattening nested ops *) \[
\begin{align*}
&[f:\text{And}|g]:\text{And} \implies \text{join}(f,g) \\
&[f:\text{Or}|g]:\text{Or} \implies \text{join}(f,g)
\end{align*}
\]

(* De Morgan’s laws *) \[
\begin{align*}
&[x:\text{Or}|\ldots]:\text{Not} \implies \\
&\color{gray}{\text{fold(}::,\text{And}(::)\text{, map}(\lambda f.\{f=f\}, x))} \\
&[x:\text{And}|\ldots]:\text{Not} \implies \\
&\color{gray}{\text{fold(}::,\text{Or}(::)\text{, map}(\lambda f.\{f=f\}, x))}
\end{align*}
\]

(* Distributivity *) \[
\begin{align*}
&[x:\text{Or}|s]:\text{And} \implies \text{map}(\lambda f.f::s, x)
\end{align*}
\]

(* Induction *) \[
\begin{align*}
&x:\text{And} \implies \text{DNF}(x) \\
&x:\text{Or} \implies \text{DNF}(x) \\
&x:\text{Not} \implies \text{DNF}(x)
\end{align*}
\]

Each rule is a straightforward translation in MGS of a well

known transformation of a boolean formula into an equivalent

one. In this program, the map and fold are the usual map

and fold functions: \(\text{map}(f,s)\) applies the function f to each

elements of the collection s and returns the collection of

results; \(\text{fold}(f,z,s)\) reduces the elements of the collection

s using the binary function f and starting from z. The function

\text{join} is used to append two collections and \text{choose} is used

to pick-up one element in a collection. Finally, the expression

\(\lambda f,f::s\) is a lambda expression that appends its argument

(here a formula) to the collection s.

B. A Simple Space Subdivisions Scheme

This example shows the building of a quadtree that par-
titions a set of points in 2D space. Quadtrees recursively
subdivide a rectangular spatial domain into four regions. The
subdivision is recursively iterated until there is less than n
points in each region (we take n = 2 in the following).
Figure 1 gives an example where the points are more or less
aligned along a curve.

This adaptive mesh is described by the following type
definitions:

type QuadTree = Grid[QuadTree] | Cloud

and gbf Grid = <n, e; 2e=0, 2n=0>

and collection Cloud = set[Point2D]

and record Point2D = { x:real, y:real }

GBF are collections with a regular neighborhood whose topol-
y is specified though a group presentation. Here, the GBF

\begin{align*}
&\text{trans } DN F = \{
\end{align*}

\begin{align*}
&\text{fun } \text{SplitCloud } (c: \text{Cloud}) =
\end{align*}

\begin{align*}
&\text{let } g = \text{barycenter } (c) \text{ in}
\end{align*}

\begin{align*}
&\text{let } c_0, c_1 = \text{split } (\lambda p.p.x<g.x, c) \text{ in}
\end{align*}

\begin{align*}
&\text{let } c_{00}, c_{01} = \text{split } (\lambda p.p.y<g.y, c_0) \text{ in}
\end{align*}

\begin{align*}
&\text{let } c_{10}, c_{11} = \text{split } (\lambda p.p.y<g.y, c_1) \text{ in}
\end{align*}

\begin{align*}
&\text{Grid: } (c_{00}@c_0, c_{01}@c.e, c_{10}@c.n, c_{11}@c.(n+e))
\end{align*}
C. Fraglet

Fraglets are tiny computation fragments or sequences of tokens that flow and react through a computer network. They have been introduced in [25] as an execution model for computer communications inspired by molecular biology. They have been designed to lay the ground for automatic network adaption and optimization processes as well as the synthesis and evolution of protocol implementations. Table I sketches the core instructions.

For example, the fraglets below together with a fraglet [length tail] located on the same node of the network, will compute the length of tail by generating the fraglet [total n] (where n is the size of tail):

[counter 0]
[matchp length empty stop cnt]
[matchp stop match counter total]
[matchp cnt pop cnt1]
[matchp cnt1 split match counter incr counter * length]
[matchp incr exch sum 1]

In this program, the fraglets can be interpreted as follows: fraglet [counter 0] defines a local variable with initial value 0; fraglets starting with matchp define functions; finally fraglet [length tail] is the application of function length on the list tail.

In the following we encode the fraglet formalism by implementing a fraglet interpreter in MGS. For the sake of simplicity, we do not consider here the localization of the fraglets on the nodes of a communication network and the communication rules between nodes. Let consider the following MGS type declaration:

```
(type Token = int | 'nul | 'exch | ...)
and collection Fraglet = seq[Token]
and collection State = bag[Fraglet]
```

The state of the system is represented by a multiset inhabited by a population of fraglets; fraglets are sequences of tokens (symbols or integers). For each fraglet operator, Table I gives the formal fraglet instruction, its informal semantics and its syntactic construction. For example, the split instruction consists in extracting the subsequence of tokens in the fraglet located between the operator (the first element in the sequence) and the first occurrence of the special token @. This operation is straightforwardly translated in MGS: the pattern matches in a fraglet the operator ‘split (the syntactic construction @ checks that the operator is located at the first position in the sequence) followed by a subsequence terminated by the special token ‘time. The subsequence is specified by (x/x != ‘time)@ that matches a repetition of elements different from ‘time.

Throughout the reader is invited to pay attention that this restriction is done for the sake of the simplicity: the whole formalism can be specified in MGS using an additional level of nesting by considering a graph labeled by multisets of fraglets.

VI. RELATED WORK

Topological collections are reminiscent of Data-fields, studied e.g. by B. Lisper [26]. Data-fields are a generalization of the array data structure where the set of indices is extended to all $\mathbb{Z}^n$ (see also [27]). We have introduced the concept of group based fields, or GBF [28], [29], to extend data-fields towards more general regular data structures. Topological collections emphasize data structures as a set of places independently of their occupation by values. This approach is also shared by the theory of species of structures [30]. Motivated by the development of enumeration techniques for labeled structures, the emphasis is put on the transport of structures along bijections while spatial computing focuses on topological relationships.

Disentangling the elements in a data structure from their organization has several advantages. In [31], B. Jay develops a concept of shape polymorphism where a data structure is also a pair (shape, set of data). The shape describes the organization of the data structure (restricted to tabular organizations) and the set of data describes the content of the data structure. This separation allows the development of shape polymorphic functions and their typing: the shape of the result of a shape polymorphic function application depends only on the shape of the argument, not of its content. The same line is developed in the field of polytypic programming for algebraic data type [32]. MGS transformations are naturally polytypic and extend far beyond arrays and algebraic data type. Polytypism in MGS relies on a generic implementation of pattern matching [24] not on overloading or ad-hoc polymorphism.

Transformations are a kind of rewriting that differs in many ways from graph rewriting. Their formalization in [33] is not based on the usual graph morphisms and pushouts like in [34] but is inspired by the approach of J.-C. Raoult [35] where graph rewriting based on a (multi-)set point of view is developed. The proposed model is close to term rewriting modulo associativity and commutativity (where the left hand side of a rule is removed and the right hand side is added). This kind of approach also allows to extend results from term rewriting to topological rewriting (as we did for termination in [36]). Note that the notions of topological collection and topological rewriting are more general than labeled graphs and graph rewriting, and may handle higher dimensional objects, a feature relevant in a lot of application areas [37].

Nested data structure are now widespread in programming languages but are less natural in the context of data bases. The importance of organizing the accesses to the element in a complex structure trough primitive operations related to the type constructor is stressed in [38]. In MGS, accesses rely on pattern matching, and the pattern matching constructs reflect the spatial structure underlying a collection. Nevertheless, structural recursion, advocated in [38], is straightforward as showed by the programs in sections IV-A and IV-B.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented the advantages of nesting in a spatial model of computation, the MGS experimental language. The language MGS, has been used in the context
of P systems [13] and in several large modeling projects in systems biology [39], [14], [40].

One interest of the spatial paradigm à la MGS is its ability to subsume several computational models in a single uniform formalism, as long as one focuses on programming [41], [42]. We showed the benefits of considering nested spatial computing through three kind of examples: in algorithmic, in simulation of multiscale phenomena and in the emulation of other programming models.

The management of nested collections is achieved through three kinds of devices:

1) collections are first-citizen values and can be used as the values of another collection;
2) a specific pattern construction [ \( p | \ldots \) ] makes possible, within the current pattern, to refer to the elements matched by a pattern \( p \) in a nested collection;
3) recursive type declarations generate predicates used to constrain the nesting and to control the pattern matching facilities.

These three features together enable a very concise and readable programming style, as exemplified in section III-D. All the presented examples are actual MGS programs, at the exception of some slight syntactic sugar.

The work presented in this paper may be enriched and extended in several directions. The pattern matching we have presented can be seen as operating at an “horizontal level” on the elements of a collection and at a “vertical level” when descending to match some elements of a nested collection. The constructions dedicated to the horizontal level are very expressive, allowing for example the matching of an unknown number of elements. The handling of the vertical level is actually restricted to the \([ \text{pat} | \ldots \) ] operator. Other construction can be designed, by analogy with the vertical level. For example, an operator to allow references through an unknown number of nesting, in a manner analogous to the iteration operator “\( \ast \)”, woould be interesting to mimic path queries in XML. Note however that the distinction between horizontal and vertical level is questionable. An alternative approach would be to unify the nested collection by looking for the spatial relationships holding in the whole structure, irrespectively of the horizontal or the vertical view. The topology of this “flat whole structure” can be built as the topology of a fiber space over the top collection. The investigation of this framework remains to be done.

ACKNOWLEDGMENTS

The authors would like to thanks H. Klaudel, F. Pommereau, F. Delaplace and J. Cohen for many questions, encouragements and sweet cookies. This research is supported in part by the ANR projects SynBioTIC.

REFERENCES
