Unconventional and Nested Computations in Spatial Computing

JEAN-LOUIS GIAVITTO[1]*, OLIVIER MICHEL[2]†, ANTOINE SPICHER[2]‡

[1] *UMR 9912 STMS, IRCAM - CNRS - UPMC - Inria, Paris, France*
[2] *LACL, Université Paris-Est Créteil, France*

Received ??? January 2012; In final form ???

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 nested topological collections in MGS, a spatial computing 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. Furthermore, we propose an extension of the MGS pattern-matching facilities to handle nesting explicitly. This makes possible the emulation of a larger class of unconventional programming models.

Key words: MGS, topological collection, transformation, topological rewriting, nested data structure, chemical computation, Lindenmayer systems, cellular automata, data field, GBF, bead sort, fraglet.

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

* email: `jean-louis.giavitto@ircam.fr`
† email: `olivier.michel@u-pec.fr`
‡ email: `antoine.spicher@u-pec.fr`

systems is not (a string representing a derivation in) another Lindenmayer system.

In *chemical computing*, as exemplified in Gamma [3], 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*[⋆] [41] 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 nested structures and their management in MGS, a spatial computing language[†] . Spatial computing 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 relationships, 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 emulation of a larger class of unconventional programming models.

Outline of the Paper. This paper is organized as follows. The next section introduces the emerging field of *spatial computing* and the notion of nesting in this context. Section 2 introduces topological collection and transformation developed in MGS, along with the required syntax to understand the examples given in the next sections. Section 3 illustrates MGS through the encoding of three simple but paradigmatic examples in unconventional computational models. We stress the fact that this encoding is useful from a programming perspective. Calculability and complexity of the underlying models are not considered in this paper. Section 4 exemplifies the use of nested spaces with

[⋆] Nested multisets are also considered in High Order Chemical Language [4]. In Structured Gamma [16], elements of the multiset are linked by relations defined by a graph grammar. It is then theoretically possible to encode a given static nest of multisets using relations specified by a specific graph grammar to implement membership test and to make a distinction between elements and nested multisets.

[†] The MGS environment can be downloaded from the MGS home page at `mgs.spatial-computing.org` where numerous other examples are developed. All the presented examples are actual MGS programs.

three direct applications. The first encodes terms used to represent boolean formulae with nested sets. The computation of a disjunctive normal form on this representation is explained. The second example computes *quadtrees*, a recursive data structure for partitioning a two dimensional space. The last one is dedicated to the informal translation of the fraglets computation model into MGS. Related and future work concludes this article.

1 MOTIVATIONS

1.1 Computing in Space, Space in Computation and Spatial Computing

Spatial Computing is an emerging field of research [12] where the computation is structured in term of *spatial relationships*: 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 systems, 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 [19]: 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 $E = \{e_1, \dots, e_n\}$ 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 simplicial complex* [22] which is a spatial representation of the interactions in the computation. 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.

As an example, consider parallel computing. Parallel computing deals with both logical and physical constraints: computations are distributed on physical distinct computing resources but the distribution of the computation is a parameter of the execution, a choice done at a logical level to minimize the computation time, and does not depend on some imposed physical localizations induced solely by the problem to be solved.

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 [1], claytronics [2], distributed robotics or programmable matter... to cite a few examples where notions like position and shape are at the core of the application domain.

1.2 Nested Spaces

A distinctive feature of spatial computing with respect to distributed computing is the emphasis put on the explicit representation of spatial relations and objects: space is seen as a data-structure on which the program is acting rather than the container of parallel computations. The direct consequence is that in spatial programming languages like Proto [7] and MGS, space is handled as *fields*. In physics, a field is the assignment of a quantity at each point of a spatial domain [34].

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. It is not common to consider "field-valued fields" in contrast to classical data structures where such a feature corresponds to nesting. Nesting is relevant and valuable in at least three situations: 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.

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 structures (*e.g.*, lists, trees) is well established. We give an example of the use of nested spaces in an algorithmic application relying on nested sets in section 4.2.

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* [8]. This methods 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 of interest. A simplistic example is presented in section 4.3. Another example, in the area of discrete modeling, is the *complex automata* framework [30] 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 4 to ease the reference between scales.

Stratified Computational Models. Some models of computation exhibit naturally an inductive structure. For instance, the state of a *membrane systems* 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 [11]. 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* [37] arrange the membranes and their interactions following an arbitrary graph, calling for a "graph of multisets". *Spatial P systems* [6] 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 symbols, multisets and grids".

In section 4.4, we will sketch the encoding of *fraglets*, a molecular bio-inspired execution model for computer communications leading to "graphs of multisets of sequences".

2 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. The MGS language represents spatial structures and their relationships through *topological collections* and relies on *transformations* for their manipulations.

MGS embeds the idea of topological collections and their transformations into the framework of a functional language. Collections are new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules.

2.1 Topological Collections

MGS handles spatial domains defined by *abstract cellular complexes* [51]. 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 the *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 [40] and have been introduced in [25] to describe arbitrary complex spatial structures that appear in biological systems [21] and other dynamical systems with a time varying structure [17, 27]. They generalize fields because they associate a quantity with 0-cells (points in space) but also with arbitrary n-cells.

Graphs (that are made of only 0- and 1-cells) are examples of one dimensional cellular complex. In this paper, we will stick with topological collections whose underlying complex is a graph. In [19] it has been showed how usual data structures can be seen as topological collections of this kind: the elements in a data structure are the quantities assigned by the field to the nodes of a graph.

We sketch here informally the collection types we use in the examples below.

Monoidal Collections. Sets, multisets and sequences are called *monoidal collections* because they can be built as a monoid with operator *join*: a sequence corresponds to a join that has no special property except associativity;

multisets are obtained with an associative and commutative join; sets when the join operator is associative, commutative and idempotent. The join operator with its properties induces the topology of the collection and the neighborhood relationship: a linear graph for sequences and a complete graph for sets and multisets.

We write `a::m` to add a value `a` in a monoidal collection `m`; the notations `seq:()`, `bag:()` and `set:()` refer to the empty sequence, the empty multiset and the empty set respectively. This must not be confused with the expression x`:set` which checks if value x is a set.

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

The topology associated with records is the "totally disconnected" ones: slots in records are isolated points (they have no neighbor).

GBF. Topological collections can be defined as *Group Based Fields* (GBF), that can be considered as associative arrays whose indices are elements in a group [28]. The latter is defined by a *finite presentation*: a set of generators together with some constraints on their combinations. Thus a GBF can be pictured as a labeled graph where the underlying graph is the Cayley graph of the finite presentation. The labels are the values associated with the vertices and the generators are associated with the edges.

For instance, in order to define a NEWS grid, we may use two generators e (east) and n (north), supporting addition, difference and multiplication by an integer. This is illustrated in the left part of Fig. 1.

Similarly, an hexagonal grid (6 neighbors for each vertex) can be defined by means of three generators n, e and nw (north-west) and a constraint $\mathsf{n} - \mathsf{nw} = \mathsf{e}$, as illustrated in the right part of Fig. 1. Notice that such graph is adequate to represent a tiling with a hexagonal shape, since the grid can be paved with hexagons centered at the positions in the grid. As shown by the dashed path, we have $2 \cdot \mathsf{n} + \mathsf{e} = 2 \cdot \mathsf{e} + \mathsf{n} + \mathsf{nw}$, which can be also checked in an algebraic way, by substituting nw with $\mathsf{n} - \mathsf{e}$ in this equality as allowed by the constraint.

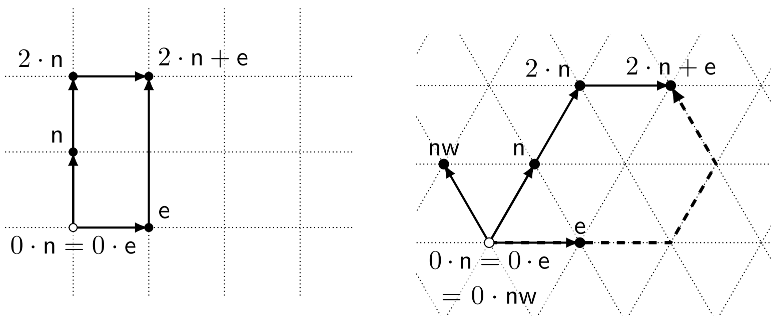

Figure 1
Left: a GBF defining a NEWS grid, with two generators e and n. Right: a GBF defining an hexagonal grid with three generators e, n and nw, and a constraint $\mathsf{n} - \mathsf{nw} = \mathsf{e}$.

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

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

2.2 Transformations

Usually in physics, fields and their evolution are specified using differential operators. MGS generalizes these operators in a rewriting mechanism, called *transformation*. A transformation is the application of some local rules following some strategy. The application of a local rule *pattern* $\Longrightarrow$ *expression* in a collection C: (1) selects a subcollection A that matches the $pattern$; (2) computes a new subcollection B as the result of the evaluation of the *expression* instantiated with the collection A; and (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. A transformation T is a function specified by a set of local rules:

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

When there is only one rule in the transformation, the enclosing braces can be dropped.

Transformations are a powerful means to define functions on topological collections complying with the underlying spatial structure. For instance, a discrete analog of differential operators can be defined using transformations [27]. For multisets, transformations reduce simply to associative-commutative rewriting [13] also called multiset rewriting.

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

$$pat := x \mid \{...\} \mid pat\texttt{,}pat' \mid pat\texttt{:}P \mid pat/exp \mid pat\ \texttt{as}\ x \mid pat\star$$

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

variable: a pattern variable x matches exactly one element. The variable x can then occur elsewhere in the rest of the rule. The construction $pat\ \texttt{as}\ x$ is used to introduce a fresh variable x that refers to the subcollection matched by pat.

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

neighbor: $pat\texttt{,}pat'$ is a pattern that matches two connected subcollections matched by pat and pat'. For example, x, y matches two connected elements (*i.e.*, y must be a neighbor of x). The connection relationship depends on the collection kind.

guard: pat/exp matches the subcollections matched by pat verifying *exp*. Pattern $pat\texttt{:}P$ is a syntactic sugar for $(pat\ \texttt{as}\ x)\ /\ P(x)$. For instance, $y\ /\ y > 3$ matches an element y provided that $y > 3$ holds and x`:int` matches an element x provided that x is an integer.

repetition: pattern $pat\star$ matches (a possibly empty) subcollection of elements matched by pat.

Rule and Application Strategy
A transformation is a set of rules. When a transformation is applied to a collection, the default strategy is to apply the first rule as many times as possible

in parallel (a rule can be applied if its pattern matches a subcollection). In the remaining collection, the second rule is applied as many times as possible in parallel with the first, and so on. This strategy is the *maximal parallel application strategy* used in *L systems* or *P systems*. Several other strategies are available in MGS like the *Gillespie application strategy* [45] based on Gillespie Stochastic Simulation Algorithm used in the modeling of chemical reactions. Strategies provide a fine control over the choice of the rules applied within a transformation. They are often non-deterministic, *i.e.*, applied on a collection, only one of the possible outcomes (randomly chosen) is returned by the transformation.

A transformation T is a function like any other function and a *first-class* value. It allows to compose transformations very easily in a higher order functional programming style. The expression $T(c)$ denotes the application of one transformation step to the collection c. A transformation step consists of the application of the rules following the rule application strategy. A transformation step can be effortlessly iterated:

$T[n]\,(c)$	denotes the application of n transformation steps to c
$T[\texttt{fix}]\,(c)$	application of the transformation T until a fixpoint

3 ENCODING THREE UNCONVENTIONAL MODELS

In this section, we illustrate the spatial approach instantiated in MGS with the encoding of three unconventional models of computation: Gamma (chemical computing) [3], L systems [44], cellular automata (CA) and related models of crystalline computing [36, 48]. We assume that the reader is familiar with the main features of these formalisms.

By "encoding", we mean that it is easy to express these unconventional programming styles using specific organizations of the underlying collections. We advocate that few notions and a single syntax can be consistently used to unify these formalisms for programming purposes.

The notion of programming style or expressiveness remains difficult to formalize: for instance, since nearly all programming languages are Turing complete, it is irrelevant to consider their set of computable functions as a measure of expressiveness. As far as we know, there are only a few attempts to formalize this notion [15, 38]. These works mainly rely on the idea of translating a language into another, using a limited and predefined form of translation (if any translation is allowed, a universal language can be the target of the translation of any other one).

The rest of this section gives some examples that show the MGS ability to mimic some paradigmatic constructions of the mentioned three computing models.

Chemical Computing. The chemical reaction model was originally proposed in 1986 as a formalism for the definition of programs without artificial sequentiality [3]. The computation proceeds by non deterministic rewriting of a multiset, until a stable state is reached. We give below the MGS version of the computation of the Fibonacci numbers in Gamma implying a change in the number of elements in the multiset:

```
fun Fib(n) = Add[fix](Dec[fix](n::bag:()))
and trans Add = { x,y ⟹ x + y }
and trans Dec = { 0 ⟹ 1;    x/(x>1) ⟹ (x-1), (x-2) }
```

The initial value n is decomposed into `1`s by the iteration of the `Dec` transformation until a fixpoint is reached. These `1`s are then summed up by transformation `Add`. The successive applications of `Dec` correspond to the recursive descent of the usual recursive definition of the Fibonacci function while the iterations of `Add` correspond the recursive ascent.

Lindenmayer Systems. An *L system* is a *parallel* string rewriting system (every production rule that might be used at each derivation state are triggered simultaneously) developed by A. Lindenmayer in the sixties [35]. It has since become a formalism used in a wide range of applications from the description of cellular interactions [35] to a model of parallel computation [42].

We will encode the seminal example of *D0L systems* in MGS. 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. The numerical argument of the L system gives the number of interactions in the rewriting process; therefore a 0L system is a context free L system (whereas an n*L system* is context sensitive with n interactions).

Lindenmayer considered the development states of a one-dimensional organism (*i.e.*, a filamentous organism) [35]: each derivation step represents 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 in dividing itself whereas the b state is a waiting state of one division step. In addition, a cell is polarized to the left or to the right which is denoted by the use of the l and r indices. The fate of each cell is specified by the rule in figure 2. A derivation tree of the process is detailed in the right of the figure. The polarity changing rules of this example are very close to those found in the blue-green bacterium *Anabaena catenula* [39, 33].

$\boldsymbol{\omega}$	$: b_r$	t_1	$: b_r$
p_1	$: a_r \to a_l\, b_r$	t_2	$: a_r$
p_2	$: a_l \to b_l\, a_r$	t_3	$: a_l\, b_r$
p_3	$: b_r \to a_r$	t_4	$: b_l\, a_r\, a_r$
p_4	$: b_l \to a_l$	t_5	$: a_l\, a_l\, b_r\, a_l\, b_r$

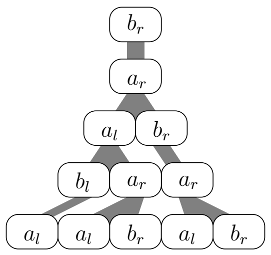

Figure 2
Model of the development of *Anabaena catenula* by a L system and the first derivations of the axiom $\boldsymbol{\omega} = b_r$.

The implementation of the production rules in MGS is straightforward. MGS symbols are used for L systems symbols. The production rules directly translates to MGS rules. So the grammar in figure 2 leads to the following MGS program:

```
trans Anabaena = { `ar ⟹ `al, `br
                   `al ⟹ `bl, `ar
                   `br ⟹ `ar
                   `bl ⟹ `al      }
```

The default application strategy of MGS is the relevant one to emulate L systems. The evaluation of *Anabaena*`[n](ar)` for n from 1 to 5 returns the sequence listed at right of figure 2. More generally, it is possible to describe the whole class of D0L systems in MGS.

Cellular Automata. To illustrate the encoding of a cellular automaton in MGS, we consider the growth of a snowflake on the hexagonal grid specified in figure 1.

Ice forms when the water is cooled below its freezing point. Crystals start from a seed and then grows by progressively adding more molecules to their surface. As an idealization, the molecules of a snowflake lie on an hexagonal grid and when a piece of ice is added to the snowflake, the heat released by this process inhibits the addition of ice nearby. This phenomenon leads to the following cellular automaton rule [52]: a black cell (value `1`) represents a place of the crystal filled with ice and a white cell (value `0`) is an empty place. A white cell becomes black if it has exactly one black neighbor, otherwise it remains white. The corresponding MGS transformation is:

```
trans SnowFlake =
   x / (NeighborFold(+, 0, x)==1) ⟹ 1
```

The construct `NeighborFold` is not a function but an operator available only within a rule: it enables to fold a function on the defined neighbors of an element matched in the lhs. Here, this operator is used to compute the number of black cells by summing the `1`s in the neighborhood (the accumulating function is the sum and the initial value is `0`). Some iterations starting from an initial grid with only one cell set to `1` are illustrated in figure 3.

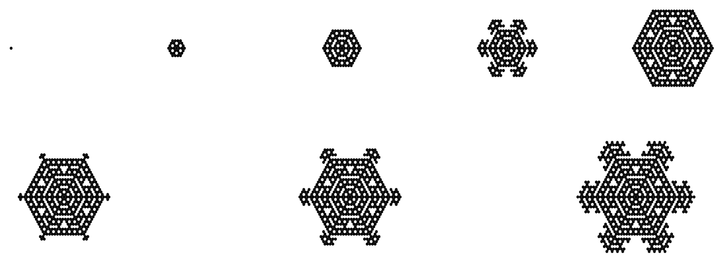

Figure 3
Formation of a snowflake. The pictured states are the step at time steps 1, 4, 8, 12, 16, 18, 20 and 23.

4 COMPUTING WITH NESTED SPACES

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:

```
fun Apply(x) = EvalRule(ApplyNested(x))
and trans ApplyNested = x:bag ⟹ Apply(x)
```

This piece of code is enough to trigger the chemical rules specified by the transformation *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.

In this section, we extend MGS to deal more easily with nested spaces, then we give examples of the three uses of nested spaces we have identified in section 1.2.

4.1 Nested Spaces in MGS

The handling of space in MGS faces two kinds of problems:

1. Often, nesting necessitates the handling of heterogeneous values (*e.g.*, in membrane systems, a membrane may contain atoms and sub-membranes). Dynamic types can be used to facilitate the management of heterogeneity.

2. The pattern constructions presented in section 2.2 are “flat”: the pattern variables of a transformation T refer to elements of the collection on which T is applied. The pattern language can be enhanced to allow a direct access to elements in nested collections.

User-Defined Subtypes. MGS is a *dynamically typed* language: no static type checking occurs but 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 patterns.

In addition to scalar values like integers, floats, strings, lambda-expressions, etc., MGS handles several kinds of collections. The elements in a collection can be of any kind, thus achieving complex objects in the sense of [10].

Often there is a need to distinguish collections of the same kind (*e.g.*, several multisets nested in another multiset). Various ways can be used to achieve the distinction. We choose to distinguish between collections of the same kind by *subtyping*. The subtype of a collection must be thought as a “color” that does not change the kind of the collection. Collection subtype declarations look like:

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

This declaration specifies a hierarchy of three subtypes. Type `MySet` is a subtype of `set`and a supertype of `AnotherMySet`. The type `AnotherSet` is also a subtype of `set` but is not comparable with `MySet`. Note that the declaration allows the restriction of the types of the elements: an `AnotherSet` only contains integers. For each type T, there is an associated predicate with the same name that can be used to check if a value has type T. For example, the expression `MySet("this is a string")` returns `false`.

No restriction takes place on type definitions: they can be recursive to handle complex nesting situations. For example, membranes containing both atomic symbols and membranes may be defined as follows:

```
collection Membrane = set[Atoms | Membrane]
and type Atoms = symbols
```

Nesting Pattern. To make easier the handling of nested collections, we extend the current pattern language with a new construction allowing references to elements of a nested collection. The pattern

```
[pat | x]
```

matches a collection C nested within the current one. The pattern pat must match a subcollection C' in C and the variable x is bound to the collection C'' obtained by removing the elements of C' from C.

For sets or multisets, C'' is the complement of C' in C. For sequences, C'' cannot be described as an interval of C. For instance, against the sequence `(0, (1, 2, 3, 4), 5)`, the pattern

```
x, ([2, 3|z] as y)
```

leads to the following unique matching:

- x is bound to `0`,
- y is bound to `(1, 2, 3, 4)`,
- z is bound to `(1, 4)`, that is, the sequence y where the sub-sequence `(2, 3)` has been removed.

Note that the second part of the nesting operator has to be a variable (it is not a pattern). The notation `[`pat` | ...]` can be used to spare a variable if the rest of the subcollection is not used elsewhere.

4.2 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:

```
collection formula = string | Not | And | Or
and record Not = { neg:formula }
and collection And = set[formula]
and collection Or = set[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 slot `neg`, or the conjunction (resp. disjunction) of formulas nested in a set with subtype `And` (resp. `Or`). With these types, the formula $\neg(p \wedge q) \vee r$ can be represented as follows:

```
{ neg = "p"::"q"::And:() }::"r"::Or:()
```

The computation of the disjunctive normal form can be achieved by iterating until a fixpoint is reached, the transformation DNF:

```
trans DNF = {
    (* Simplifying unaries *)
    [[x|...]:Not|...]:Not ⟹ x        (* ¬¬x = x *)
    x:And / size(x)=1 ⟹ hd(x)
    x:Or / size(x)=1 ⟹ hd(x)
    (* Flattening nested ops *)
    [f:And|g]:And ⟹ join(f,g)
    [f:Or|g]:Or ⟹ join(f,g)
    (* De Morgan's laws *)
    [x:Or|...]:Not ⟹ fold(::, And:(), map(Negate,x))
    [x:And|...]:Not ⟹ fold(::, Or:(), map(Negate,x))
    (* Distributivity *)
    [x:Or|s]:And ⟹ map(λy. y::s, x)
    (* Induction *)
    x:And ⟹ DNF(x)
    x:Or ⟹ DNF(x)
    x:Not ⟹ DNF(x)
}

fun Negate(x) = { neg = x }

fun Dnf(x) = if x:Or then DNF[fix](x)
             else DNF[fix](x::Or:())
```

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: `map(`f`,`s`)` applies the function f to each elements of the collection s and returns the collection of results; `fold(`f`,`z`,`s`)` reduces the elements of the collection s using the binary function f and starting from z. The function `join` is used to join two monoidal collections and `hd` is used to pick-up one element in a collection. The expression $\lambda y.\ y::s$ is a lambda expression that appends its argument to the collection s.

Named functions are introduced with the `fun` keyword. The function Dnf ensures that the formula to normalize is a conjunction and then iterates the transformation DNF until a fixpoint. The transformation DNF itself is recursive, but on the substructure of its argument: the recursion of a transformation is always primitive because the rules are applied on proper parts of the argument.

4.3 A Simple Space Subdivisions Scheme

This example shows the building of a quadtree that partitions 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 4 gives an example.

This adaptive mesh is described by the following type definitions:

```
collection quadTree = G22[quadTree | cloud]
and gbf G22 = < n, s, e, w ; n+s=0, w+e=0, 2e=0, 2n=0 >
and collection cloud = set[point2D]
and record point2D = { x:float, y:float }
```

The GBF `G22` specifies a 2×2 torus with two directions n and e with inverses s and w respectively. The recursive subdivision is computed by the transformation $MakeQuadTree$:

```
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
        G22:(c00@0, c01@e, c10@n, c11@(n+e))
```

Function $SplitCloud$ divides a cloud of points `c` into four subclouds `c`ij 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 `G22` collection where the four cells are labeled by the clouds `c`ij. 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:(`$\ldots v_i$`@`$c_i \ldots$`)` builds a collection of type `T` where the value v_i is associated with the cell c_i. The process is illustrated on figure 4.

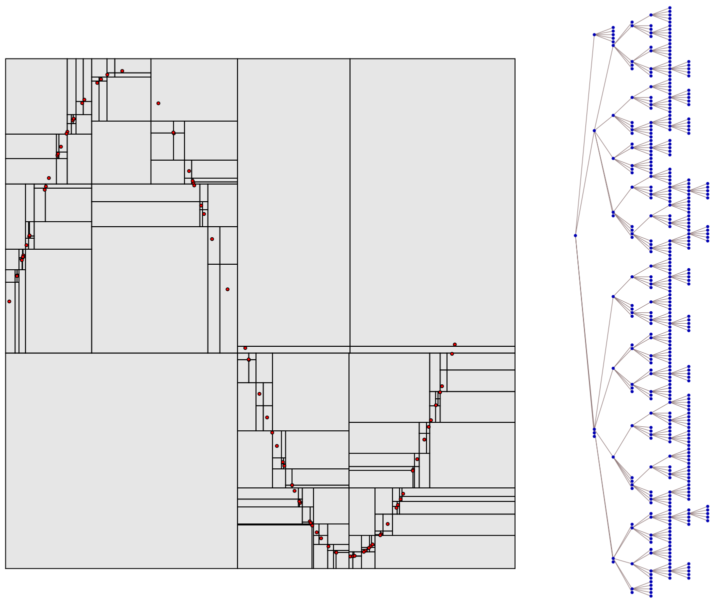

Figure 4
Left: Adaptive mesh refinement using quadtrees on a set of 100 points. *Right:* The corresponding nested structure pictured as an inclusion tree.

4.4 Fraglet

Fraglets are *active messages*: tiny computation fragments that flow and react through a computer network. Each fraglet is a sequence of symbols, some of them representing actions, some of them representing data. Following its head symbol, a fraglet may evolve to restructure the rest of its content or interact with another fraglet presenting some specific structure. The result of the interaction can be the creation of new fraglets or the transmission of the reactants into another part of the network.

Fraglets have been introduced in [50] 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 1 sketches the core instructions.

For example the six following fraglets

```
(1)   [counter 0]
(2)   [matchp length empty stop cnt]
(3)   [matchp stop match counter total]
(4)   [matchp cnt pop cnt1]
(5)   [matchp cnt1 split match counter incr counter × length]
(6)   [matchp incr exch sum 1]
```

located together with a fraglet `[length` *`tail`*`]` 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`*). 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`*.

We detail a simple run: on the fraglet `[length a]` only fraglet (2) apply and the former is replaced by `[empty stop cnt a]` which self-evaluates to `[cnt a]`. Fraglet (4) reacts and the result is `[pop cnt1 a]` which in turn results in `[cnt1]`. This last one interacts with (5) and gives `[split match counter incr counter × length]`. It self-evaluates into the two fraglets `[match counter incr counter]` and `[length]`. The former reacts with (1) and the two are replaced by `[counter 1]`. The latter reacts with (2) and produces `[empty stop cnt]` which reduces to `[stop]`. Then, with (3), the fraglet `[match counter total]` is produced and combines with `[counter 1]` to gives the final `[total 1]`. No remaining fraglets may interact furthermore.

MGS encoding. 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 inst = int | 'nul | 'exch | 'matchp | ...
and collection fraglet = seq[inst]
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 1 gives the fraglet instruction, its informal semantics and its translation into an MGS transformation rule. These rules are part of the transformation $EvalFraglets$:

[‡] Nevertheless the reader is invited to pay attention that this restriction is done to keep the presentation simple: the whole formalism can be specified in MGS by using an additional level of nesting considering a graph labeled by multisets of fraglets.

Op	*Input*	*Output*
nul	[nul *tail*] destroy a fraglet [‘null@0 | tail]:fraglet $\Longrightarrow$ fraglet:()	[]
dup	[dup t a *tail*] duplicate a single symbol [‘dup@0,t,a | tail]:fraglet $\Longrightarrow$ t::a::a::tail	[t a a *tail*]
exch	[exch t a b *tail*] swap two tags [‘exch@0,t,a,b | tail]:fraglet $\Longrightarrow$ t::b::a::tail	[t b a *tail*]
split	[split s1 $\times$ s2] break a fraglet into two at the first occurrence of $\times$ [‘split@0,(x/x!=‘time)* as s1,‘time | s2]:fraglet $\Longrightarrow$ s1, s2	[s1] [s2]
fork	[fork a b *tail*] copy a fraglet and prepend different header symbols [‘fork@0,a,b | tail]:fraglet $\Longrightarrow$ a::tail, b::tail	[a *tail*] [b *tail*]
pop	[pop h a *tail*] pop the “head” element of the list “a, *tail*” [‘pop@0,h,a | tail]:fraglet $\Longrightarrow$ h::tail	[h *tail*]
empty	[empty yes no *tail*] test for empty *tail* [‘empty@0,y,n | tail]:fraglet $\Longrightarrow$ if size(tail)==0 then y:: fraglet:() else n::tail	[yes] *or* [no *tail*]
sum	[sum t n_1 n_2 *tail*] arithmetic addition [‘sum@0,t,n1,n2 | tail]:fraglet $\Longrightarrow$ t::(n1+n2)::tail	[t $(n_1 + n_2)$ *tail*]
match	[match a *tail1*],[a *tail2*] two fraglets react, their tails are concatenated [‘matchp@0,a | t1]:fraglet, [b@0 | t2]:fraglet / a = b $\Longrightarrow$ join(t1,t2)	[*tail1* *tail2*]
matchP	[matchP a *tail1*],[a *tail2*] idem as match but the rule persists [‘matchp@0,a | t1]:fraglet as f, [b@0 | t2]:fraglet / a = b $\Longrightarrow$ f, join(t1,t2)	[*tail1* *tail2*]

Table 1
Subset of the fraglets core instructions (from [50]) and their MGS translations.

```
trans EvalFraglets = {
    ...rules from table 1...
}
```

which implements the evaluator of the MGS fraglet interpreter. Note that this evaluator is very simple: one rule implements one fraglet instruction.

For example, the split instruction consists in extracting the subsequence of tokens in the fraglet located between the operator (first element) and the first occurrence of the special token $\times$ (the symbol `` `times `` in MGS). This operation is straightforwardly translated in MGS: the pattern matches in a fraglet the operator `` `split `` (the syntactic construction `@0` 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 ``.

5 RELATED AND FUTURE WORK

In this paper, we have presented the advantages of nesting in a spatial model of computation, the MGS experimental language. The MGS language, has been used in the context of P systems [25] and in several large modeling projects in systems biology [5, 21, 47].

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, cf. section 3. 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:

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

Together these features enable a very concise and readable programming style, as exemplified in section 4.

5.1 Related Work

Topological collections are reminiscent of *Data Fields* studied *e.g.*, by B. Lisper [29]. Data fields are a generalization of the array data structure where the set of indices is extended to all $\mathbb{Z}^n$ (see also [18]). We have introduced the concept of *group based fields*, or GBF [23, 24], to extend data fields towards more general regular data structures. Topological collections include GBF and also irregular data structures. They share with data fields the emphasis on data structure as a set of *places* independently of their occupation by values. This approach is also shared by the theory of *species of structures* [9]. 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], C. 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 [20] not on overloading.

Nested data structure are now widespread in programming languages. The importance of organizing the accesses to the element in a complex structure trough primitive operations related to the type constructor is stressed in [10]. In MGS, accesses rely on pattern matching, and the pattern matching constructs reflect the spatial structure underlying a collection. Structural recursion, advocated in [10], is still possible, as showed by the programs in sections 4.2 and 4.3.

Transformations are a kind of rewriting that differs in many ways from graph rewriting. Their formalization in [46] is not based on the usual graph morphisms and pushouts like in [14] but is inspired by the approach of J.-C. Raoult [43] where graph rewriting based on a multiset 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 [26]).

Note that the notions of topological collection and topological rewriting are more general and may handle higher dimensional objects, a feature relevant in a lot of application areas [49].

5.2 Perspectives

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 `[` pat `|` ... `]` operator. Other vertical construction can be designed, in parallelism with the horizontal level. For example, an operator to allow references through an unknown number of nesting, in a manner analog to the iteration operator `*`, would be interesting to mimic path queries in XML. Note however that the distinction between horizontal and vertical levels is questionable. Another 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 build as the topology of a fiber space over the top collection. The investigation of this framework remains to be done.

6 ACKNOWLEDGMENTS

The authors would like to thanks the anonymous reviewers for their comments on a first version of this paper. We are grateful to 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

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T.F. Knight Jr, R. Nagpal, E. Rauch, G.J. Sussman, and R. Weiss. (2000). Amorphous computing. *Communications of the ACM*, 43(5):74–82.

[2] B. Aksak, P. S. Bhat, J. Campbell, M. De Rosa, S. Funiak, P. B. Gibbons, S. C. Goldstein, *et al*. (2005). Claytronics: highly scalable communications, sensing, and actuation networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, SenSys 2005, San Diego, California, USA, November 2-4, 2005*, page 299. ACM.

[3] J.P. Banâtre, P. Fradet, and D. Le Métayer. (2001). Gamma and the chemical reaction model: Fifteen years after. In *Proc. of the Workshop on Multiset processing: mathematical, computer science, and molecular computing points of view*, LNCS 2235, pages 17–44.

[4] J.P. Banâtre, P. Fradet, and Y. Radenac. (2007). Programming self-organizing systems with the higher-order chemical language. *International Journal of Unconventional Computing*, 3(3):161.

[5] P. Barbier de Reuille, I. Bohn-Courseau, K. Ljung, H. Morin, N. Carraro, C. Godin, and J. Traas. (2006). Computer simulations reveal properties of the cell-cell signaling network at the shoot apex in Arabidopsis. *PNAS*, 103(5):1627–1632.

[6] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, G. Pardini, and L. Tesei. (2011). Spatial p systems. *Natural Computing*, 10(1):3–16.

[7] J. Beal and J. Bachrach. (2006). Infrastructure for engineered emergence on sensor/actuator networks. *Intelligent Systems, IEEE*, 21(2):10–19.

[8] M.J. Berger and J. Oliger. (1984). Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics*, 53(3):484–512.

[9] F. Bergeron, G. Labelle, and P. Leroux. (1997). *Combinatorial species and tree-like structures*, volume 67 of *Encyclopedia of mathematics and its applications*. Cambridge University Press. isbn 0-521-57323-8.

[10] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. (18 September 1995). Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48.

[11] J. Castellanos, G. Paun, and A. Rodriguez-Paton. (2000). Computing with membranes: P systems with worm-objects. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 65–74. IEEE.

[12] A. De Hon, J.-L.Giavitto, and F. Gruau, editors. (3-8 sptember 2006). *Computing Media and Languages for Space-Oriented Computation*, number 06361 in Dagsthul Seminar Proceedings. Dagsthul, `http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=2006361`.

[13] N. Dershowitz, J. Hsiang, N.A. Josephson, and D.A. Plaisted. (1983). Associative-commutative rewriting. In *Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 2*, pages 940–944. Morgan Kaufmann Publishers Inc.

[14] H. Ehrig, M. Pfender, and H. J. Schneider. (1973). Graph-grammars: An algebraic approach. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)-Volume 00*, pages 167–180. IEEE Computer Society.

[15] M. Felleisen. (December 1991). On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75.

[16] P. Fradet and D. Le Métayer. (1998). Structured gamma. *Science of Computer Programming*, 31(2-3):263–289.

[17] J.-L. Giavitto. (June 2003). Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *14th International Conference on Rewriting Technics and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 208–233, Valencia. Springer.

[18] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet. (September 1998). A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$. In *EuroPar'98 Parallel Processing*, volume 1470 of *LNCS*, pages 742–745.

[19] J.-L. Giavitto and O. Michel. (October 2002). Data structure as topological spaces. In *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, volume 2509, pages 137–150, Himeji, Japan. LNCS.

[20] J.-L. Giavitto and O. Michel. (October 2002). Pattern-matching and rewriting rules for group indexed data structures. In *ACM Sigplan Workshop RULE'02*, pages 55–66, Pittsburgh. ACM.

[21] J.-L. Giavitto and O. Michel. (2003). Modeling the topological organization of cellular processes. *BioSystems*, 70:149–163.

[22] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher. (September 2005). Computation in space and space in computation. In *Unconventional Programming Paradigms (UPP'04)*, volume 3566 of *LNCS*, pages 137–152, Le Mont Saint-Michel. Spinger.

[23] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. (2–4 October 1995). Group based fields. In *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *LNCS*, pages 209–215, Beaune (France). Springer-Verlag.

[24] J.-L.Giavitto, (May 1998). Rapport scientifique en vue d'obtenir l'habilitation à diriger des recherches.

[25] J.-L.Giavitto and O. Michel. (2002). The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129.

[26] J.-L.Giavitto, O. Michel, and A. Spicher. (november 2008). *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, chapter Spatial Organization of the Chemical Paradigm and the Specification of Autonomic Systems, pages 235–254. Springer.

[27] J.-L.Giavitto and A. Spicher. (jully 2008). Topological rewriting and the geometrization of programming. *Physica D*, 237(9):1302–1314.

[28] J.L. Giavitto and O. Michel. (2001). Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 150–161. ACM.

[29] P. Hammarlund and B. Lisper. (June 1993). On the relation between functional and data parallel programming languages. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 210–219. ACM.

[30] A. Hoekstra, E. Lorenz, J.L. Falcone, and B. Chopard. (2007). Towards a complex automata framework for multi-scale modeling: Formalism and the scale separation map. *Computational Science–ICCS 2007*, pages 922–930.

[31] C. B. Jay. (1995). A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283.

[32] J. Jeuring and P. Jansson. (1996). Polytypic programming. In *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 68–114. Springer.

[33] C. G. Koster and A. Lindenmayer. (1987). Discrete and continuous models for heterocyst differentiation in growing filaments of blue-green bacteria. *Acta Biotheoretica*, 36:249–273.

[34] G. T. Leavens. (May 1994). Fields in physics are like curried functions or physics for functional programmers. Technical Report TR94-06b, Iowa State University, Department of Computer Science.

[35] A. Lindenmayer. (1968). Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315.

[36] N. Margolus. (April 1998). Crystalline computing. In *Proc. Conf. on High Speed Computing LANL * LLNL*, pages 249–255, Gleneden Beach, OR. LANL. LA-13474-C Conference, UC-705.

[37] C. Martín-Vide, G. Paun, J. Pazos, and A. Rodríguez-Patón. (2003). Tissue P systems. *Theoretical Computer Science*, 296(2):295–326.

[38] J. C. Mitchell. (1993). On abstraction and the expressive power of programming languages. In *TACS'91: Selected papers of the conference on Theoretical aspects of computer software*, pages 141–163, Amsterdam, The Netherlands, The Netherlands. Elsevier Science Publishers B. V.

[39] G. J. Mitchinson and M. Wilcox. (1972). Rule governing cell division in anaeba. *Nature*, 239:110–11.

[40] J. Munkres. (1984). *Elements of Algebraic Topology*. Addison-Wesley.

[41] G. Paun. (2000). Computing with membranes. *Journal of Computer and System Sciences*, 1(61):108–143.

[42] P. Prusinkiewicz and J. Hanan. (February 1992). L systems: from formalism to programming languages. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 193–211. Springer Verlag.

[43] J.-C. Raoult and F. Voisin. (1994). Set-theoretic graph rewriting. In *Proceedings of the International Workshop on Graph Transformations in Computer Science*, pages 312–325, London, UK. Springer-Verlag.

[44] G. Ronzenberg and A. Salomaa, editors. (February 1992). *L systems: from formalism to programming languages*. Springer Verlag.

[45] A. Spicher, O. Michel, Mikolaj Cieslak, J.-L.Giavitto, and Przemyslaw Prusinkiewicz. (March 2008). Stochastic p systems and the simulation of biochemical processes with dynamic compartments. *BioSystems*, 91(3):458–472.

[46] A. Spicher, O. Michel, and J.-L.Giavitto. (September 2010). Declarative mesh subdivision using topological rewriting in mgs. In *Int. Conf. on Graph Transformations (ICGT) 2010*, volume 6372 of *LNCS*, pages 298–313.

[47] A. Spicher, O. Michel, and J.-L.Giavitto. (February 2011). *Understanding the Dynamics of Biological Systems: Lessons Learned from Integrative Systems Biology*, chapter Interaction-Based Simulations for Integrative Spatial Systems Biology. Springer Verlag.

[48] T. Toffoli. (2004). A pedestrian's introduction to spacetime crystallography. *IBM Journal of Research and Development*, 48(1):13–30.

[49] E. Tonti. (Jan. 1972). On the mathematical structure of a large class of physicial theories. *Rendidiconti della Academia Nazionale dei Lincei*, 52(fasc. 1):48–56. Scienze fisiche, matematiche et naturali, Serie VIII.

[50] C. Tschudin. (2003). Fraglets-a metabolistic execution model for communication protocols. In *Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park, USA*, pages 1–6.

[51] A.W. Tucker. (1933). An abstract approach to manifolds. *The Annals of Mathematics*, 34(2):191–243.

[52] S. Wolfram. (2002). *A new kind of science*. Wolfram Media.