A formal model based on affinity among elements for describing behavior of complex systems

Kazuto Tominaga

Department of Computer Science
University of Illinois at Urbana-Champaign

Report No. UIUCDCS-R-2004-2413
March 2004

Abstract

The field of studies on complex systems is becoming one of the most active research areas in computer science. Among those systems, there is a type of system that has the following characteristics: comprising a large number of unintelligent elements, composition and decomposition of groups of elements, and simple but various interactions among the elements.

In this paper, we present a formal model for describing the behavior of such type of complex system. The presented model is based on pattern matching, recombination of elements, and nondeterminism. We give example descriptions for several systems, which include a simple self-replicating creature, Turing machines, and a solver of the 3SAT problem.

Keywords: complex systems, formal model, pattern matching, artificial life, nondeterministic computation, molecular computation.

1 Introduction

Since their emergence, studies on complex systems have been vigorously carried out. These systems are characterized by enormous amounts of elements in a system and intricate interactions among the elements. This category encompasses a wide range of systems from a group of interacting molecules to a human society.

To capture the nature of complex systems, various mathematical models have been designed and applied to describe those systems. One possible approach would be the application of differential equations to systems whose behavior can be approximated by continuous equations. On the other hand, characteristics of other systems are often difficult to express as mathematical formulae because of their discrete behavior.

Among those systems, we focus on systems that have the following characteristics.

- The system includes huge amount of distinct objects,
- each object can interact with (possibly) any other objects,
- those interactions may cause objects to stick to or part from each other,
- but each object is not "intelligent", i.e., it does not control behavior of itself,

- and the interactions are simple but of various kinds.

An example of such a system is a system of molecules contained in solution in a test tube. The molecules may interact with each other forming new molecules or decomposing, but the collection of atoms contained in the test tube never changes by those interactions. Let us call such systems *element-conserving dynamic systems*.

In the following sections, we will present a simple formal model that can describe behavior of element-conserving dynamic systems. The formal model is based on pattern matching, recombination of elements, and nondeterminism.

2 Modelling element-conserving dynamic systems

First, we set up a framework for describing element-conserving dynamic systems, in which a formal model has the following components.

- A finite multiset of objects. Each object is composed of elements. An object has some structure such as string, tree, etc.
- Finite sources of objects. Each source can generate an infinite number of objects of one type, one at a time.
- Finite drains of objects. Each drain can dispose of objects of one type, one at a time.
- A finite set of recombination rules. A rule transforms (composes and/or decomposes) one or more objects to one or more objects. The collection of elements contained in the objects before recombination is conserved by the recombination.

The main difference between recombination rules and rewriting rules of term-rewriting systems (and similar rewriting systems) is that recombination rules conserve elements when they are applied to objects, whereas term-rewriting rules can usually replace a term with one comprising arbitrarily different elements from those in the original term. In this framework, therefore, the aggregate number of elements contained in the system can only be changed by the operations of sources and drains.

In the initial state, the multiset can contain a finite number of objects. Then the sources and the drains operate, and the recombination rules are applied to objects.[1]

3 A simple formal model based on pattern matching

In this section, we present a simple formal model in the framework that can describe a class of element-conserving dynamic systems. The formal model uses pattern matching in its recombination rules.

3.1 Components of description

The following components are specified in this model to express a system. Let us suppose we are going to describe a system comprising objects that are composed of lower-case letters.

Element types. The system has a finite set of element types and each *element* is of exactly one element type. Let the lower-case alphabet (`a`, `b`, ..., `z`) denote the element types.

[1]This framework does not specify the order or timing of the effects of sources, drains and recombination rules since doing so may impair the ability of the framework to capture interesting properties of systems.

Objects. In this model, an object is a compound of one or more lines of elements, placed top to bottom adjacently like the following example.

```
 banana
peach
   apple
```

This object is composed of three lines.

As shown, each line can start at a different column from the others. To save space, we will introduce a string format to represent objects. An object is denoted by a string of elements with the starting positions of all the lines. The object above is expressed as `0:banana/-1:peach/2:apple/`; the numbers -1 and 2 represent the displacements of the second and the third line, respectively, relative to the first line.[2]

Patterns. A pattern matches (or does not match) an object. A pattern is also a compound of one or more lines placed top to bottom, and each line is a sequence of elementary patterns. There are three elementary patterns:

- Element type (i.e., a lower-case letter): This matches an element of this type. Let us call this elementary pattern *letter*.
- Wildcard for one element: This matches any single element. Let the digits (`0`, `1`, ..., `9`) denote wildcards, and call this type of wildcard *digit*. Each digit represents the matched element. This association is used in recombination.
- Wildcard that matches a sequence of zero or more elements. Let us call it *elastic*, and describe it by a pair of a digit and an asterisk (e.g., `*1` or `1*`). An elastic represents the matched sequence of elements.

We impose a restriction on the use of elastics. An elastic cannot be placed between letters or digits, i.e., an elastic must be on the left or right end of a line[3]. A preceding asterisk indicates that the elastic is at the left end of the line (e.g., `*3`), and a succeeding asterisk means the elastic is at the right end of the line (`2*`). An example pattern is as follows.

```
 1an2*
*3ach
  ap4*
```

This pattern matches the example object above. The digit `1` matches the element `b` in the object, the elastic `2*` matches the sequence of elements `ana`, and so forth. We adopt the string format for objects to express patterns. Line positions are calculated according to the first letter or digit (i.e., excluding elastics). The pattern above is denoted by `0:1an2*/1:*3ach/2:ap4*/`.

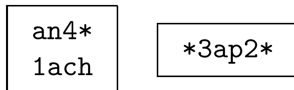
Recombination rules. A recombination rule is composed of two parts, namely, a set of *pre-recombination patterns* and a set of *post-recombination patterns*. Objects that match the pre-recombination patterns are transformed to objects expressed by the post-recombination patterns. Every recombination rule must be specified so that the multiset of all the entities contained in the objects to which the recombination rule is applied is the same as the multiset of all the entities in the resulting objects. In

[2]The displacement for the first line is meaningless, but for readability and consistency, 0 is always used.

[3]This restriction is somewhat arbitrary, but it is useful to preclude problems that might arise when an elastic should "stretch" or "shrink" due to the effect of recombinations.

other words, every recombination rule must use all and only the components of the original objects to make the new objects.

Elements represented by digits or elastics in the pre-recombination patterns are placed where the same digits or elastics appear in the post-recombination patterns. Let us consider a rule with the pre-recombination patterns including only one pattern shown above, and the post-recombination patterns being the following two patterns.

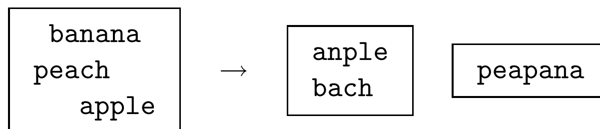


This rule is denoted by

{ `0:1an2*/1:*3ach/2:ap4*/` } → { `0:an4*/0:1ach/`,`0:*3ap2*/` }.

In this notation, patterns are separated by commas if multiple patterns are included in the pre- or post-recombination patterns, as in the post-recombination patterns shown above.

When this rule is applied to the example object shown above (`0:banana/-1:peach/2:apple/`), two objects are obtained as the result of the recombination.



Note that the collection of elements is conserved.

We restrict the use of elastics in recombination rules to keep the model simple: an elastic that is at the left end of a line in the pre-recombination patterns must appear at the left end of one of the lines in the post-recombination patterns; the same restriction applies to an elastic at the right end. In the example recombination rule, the elastic `2*`, which is at the right end of the line `an2*` in the pre-recombination pattern, appears at the right end of the line `*3ap2*`, so this usage satisfies the restriction, as well as the other elastics in the example.

3.2 Interpretation

Interpretation of a described system should be nondeterministic (that is, all possible execution sequences from the initial state are considered) since there are multiple choices for possible state changes of the system; they include generation of a new object (or objects) from one (or more) of the sources, disposing of one or more objects by the drains, recombination of objects by one or more recombination rules.

Also, operations of sources, drains and applications of recombination rules can occur in any order. But the condition that the whole element entities in the system must be conserved (changed only by the sources and the drains) should be held, so if conflicts of multiple simultaneous recombinations and/or draining of objects are to occur, they should be reconciled in accordance with this condition.

4 Example descriptions

In this section, we illustrate some usage of the formal model given in the previous section by several example descriptions.

4.1 Sentence generation

The following is the description of a system that produces sequences with any number of `ab` (i.e., `ab`, `abab`, ... , etc).

- Initial multiset: `0:cd/`.
- Sources: `0:ab/`.
- Rules:
  { `0:*1ab/`, `0:cd/` } → { `0:*1ab/1:cd/` }
  { `0:*1ab/1:cd/`, `0:ab2*/` } → { `0:*1abab2*/1:cd/` }
  { `0:*1ba2*/0:cd/` } → { `0:*1ba2*/`, `0:cd/` }

(No drain is defined.)

The source supplies unlimited number of `0:ab/` to the multiset. The first rule attaches `0:cd/` to an object ending with `ab`; the second rule attaches another object beginning with `ab` to it; then the third rule detaches `0:cd/` from the result.

4.2 Turing machines

Suppose the specification of a Turing machine [9] is given:

- $M = \{q_1, q_2, \ldots, q_R\}$: set of the states of the machine (m-configurations). Assume q_1 to be the initial machine state.
- $S = \{s_1, s_2, \ldots, s_n\}$: set of the tape symbols.
- T: transition table, each entry of which is $\langle q_i, s_k, q_j, s_l, h\rangle$, where q_i is the current state of the machine, s_k is the symbol read from the tape, s_l is the symbol to write on the tape, q_j is the next state, and h is the movement of the machine relative to the tape, which is one of { L, R, N } corresponding to left, right and no move, respectively.

Then we can construct a system as follows.

- Element types: { `a`, `m`, `t` } in addition to the following:
  - For the machine states, let each of $q_1, q_2, \ldots, q_R$ be an element type.
  - For the tape symbols, let each of $s_1, s_2, \ldots, s_n$ be an element type, and so be s_0, which denotes a blank square on the tape.
- Initial multiset: `0:`s_0`/-1:amt/0:`q_1`/`, `0:`q_1`/`, and two `0:`q_i`/`s for each $2 \leq i \leq R$. `amt` is the machine and it scans the tape square on top of `m` (the tape initially has only one square which is blank[4]). Below `m` is the current machine state (now the initial state q_1).
- Sources: `0:`s_i`/` for each $0 \leq i \leq n$, supplying unlimited number of the tape symbols.
- Rules: For each $\langle q_i, s_k, q_j, s_l, h\rangle \in T$, let this system have one rule according to h as follows.
  - $h = L$:
    { `0:*1`s_k`2*/-1:amt/0:`q_i`/`, `0:`q_j`/`, `0:`s_l`/` } → { `0:*1`s_l`2*/-2:amt/-1:`q_j`/`, `0:`q_i`/`, `0:`s_k`/` }
  - $h = R$:
    { `0:*1`s_k`2*/-1:amt/0:`q_i`/`, `0:`q_j`/`, `0:`s_l`/` } → { `0:*1`s_l`2*/0:amt/1:`q_j`/`, `0:`q_i`/`, `0:`s_k`/` }

[4]If the initial tape is not blank, it can be given by replacing `0:`s_0`/` on top of the machine with the sequence of the symbols on the tape. However, beginning with a blank tape is equivalent to giving an initial tape from the viewpoint of computational power [9], we describe the system as in the main text assuming the original Turing machine has a blank tape at its initial state.

- $h = N$:
  { `0:*1`s_k`2*/-1:amt/0:`q_i`/`, `0:`q_j`/`, `0:`s_l`/` } → { `0:*1`s_l`2*/-1:amt/0:`q_j`/`, `0:`q_i`/`, `0:`s_k`/` }

The following are rules that add blank squares to each end of the tape when it comes above `m`.*

- { `0:*12/0:amt/1:3/`, `0:`s_0`/` } → { `0:*12`s_0`/0:amt/1:3/` }
- { `0:12*/-2:amt/-1:3/`, `0:`s_0`/` } → { `0:`s_0`12*/-1:amt/0:3/` }

If the given Turing machine is deterministic, at most one rule can be chosen at any time, which is specified by the symbol s_k above `m` and the state q_i below `m`. Any object that represents the succeeding state (q_j) is always available in the multiset (each state has two objects; they enable transition to the same state), and the sources eventually produce the necessary symbol to write on the tape (s_l), so this system simulates the behavior of the original Turing machine.

Adding the following sources makes the system describe a nondeterministic Turing machine.

- a source of `0:`s_0`/-1:amt/0:`q_1`/`, supplying the machines.
- a source of `0:`q_i`/` for each $0 \leq i \leq R$, supplying the machine states.

In this case, the initial multiset can be empty.

4.3 A self-replicating creature

We can construct a very simple system that imitates a creature that replicates itself. This creature consists of its body (let us call it *a chromosome*) and a mediator that helps reproduction (let us call it *an enzyme*). A chromosome looks like `pabbabq`, where `p` designates the beginning of the chromosome, `q` the end of it, and `abbab` is the information that the chromosome carries. An enzyme is `st`. One individual of the creature is a pair of one chromosome and one enzyme.

- Initial multiset: `0:pabbabq/` and `0:st/`.
- Sources: `0:a/`, `0:b/`, `0:p/`, `0:q/`, `0:s/` and `0:t/`.
- Rules:
  { `0:p12*/`, `0:st/` } → { `0:p12*/-1:st/` }
  { `0:p1*/-1:st/`, `0:p/` } → { `0:p1*/0:st/0:p/` }
  { `0:*1a2*/-1:st/0:*3/`[5], `0:a/` } → { `0:*1a2*/0:st/0:*3a/` }
  { `0:*1b2*/-1:st/0:*3/`, `0:b/` } → { `0:*1b2*/0:st/0:*3b/` }
  { `0:*1q/-1:st/0:*2/`, `0:q/`, `0:s/` } → { `0:*1q/`, `0:st/0:s/`, `0:*2q/` }
  { `0:st/0:s/`, `0:t/`, } → { `0:st/`, `0:st/` }

The first two rules start copying a chromosome; the third and fourth rules add the corresponding base (`a` or `b`) to the new chromosome; the last two rules detach the chromosomes from the enzyme as well as producing a new enzyme.

Although this system has no concept of inside (or outside) of an individual creature, it can be regarded as imitating the reproduction system of creatures in the sense that a pair of a chromosome and an enzyme is produced at a time.

*Corrected (June 2004): In the original report, `0:`s_0`/` in the left-hand sides were missing, which was incorrect.

[5]The displacement 0 in the line `0:*3/` means the rightmost element included in `*3` (if any) is at the relative position −1. For a line composed of only a right-hand side elastic, say `0:3*/`, the leftmost element in `3*` is aligned at the position 0.

4.4 Solving 3SAT

We will illustrate an example application of the formal model to the traditional computation problem: 3SAT (satisfiability problem). Suppose we have a function f of the four variables x_1, x_2, x_3 and x_4:

$$f = t_1 t_2 t_3 = (x_1 + x_2 + x_3')(x_1' + x_3 + x_4)(x_2 + x_3 + x_4')$$

where x_i' denotes the negation of x_i.

The system described below will find an assignment that satisfies f, by looking for assignments that satisfy each of t_1, t_2 and t_3 in this order.

Initial multiset. The initial multiset is empty.

Sources.

- Sources that produce assignments that make t_1 true:
  `0:aptdddz/`, `0:apdtddz/` and `0:apddfdz/`.
  `t` means true, `f` false, and `d` don't-care; `ap` is the prefix and `z` is the terminator.
- Sources that produce assignments that make t_2 true:
  `0:bpfdddz/`, `0:bpddtdz/` and `0:bpdddtz/`.
- Sources that produce assignments that make t_3 true:
  `0:cpdtddz/`, `0:cpddtdz/` and `0:cpdddfz/`.
- Sources of the mediators: `0:as/`, `0:bs/`, `0:cs/`.
  These mediators create intermediate results of the form
  - `0:aq`$x_1x_2x_3x_4$`z/` (for t_1 satisfied),
  - `0:bq`$x_1x_2x_3x_4$`z/` (for t_1 and t_2 satisfied) and
  - `0:cq`$x_1x_2x_3x_4$`z/` (for f satisfied, which is a final result)

  where x_i is one of `t`, `f` and `d`, meaning the value of x_i is true, false and don't-care, respectively.
- Sources of parts to construct intermediate (and final) results: `0:a/`, `0:b/`, `0:c/`, `0:q/`, `0:z/`, `0:t/`, `0:f/` and `0:d/`.

Rules. For creating intermediate results that satisfy t_1:

{ `0:ap1*/`, `0:as/`, `0:a/`, `0:q/` } → { `0:ap1*/1:as/0:aq/` }
{ `0:*1t2*/-1:as/0:*3/`, `0:t/` } → { `0:*1t2*/0:as/0:*3t/` }
{ `0:*1f2*/-1:as/0:*3/`, `0:f/` } → { `0:*1f2*/0:as/0:*3f/` }
{ `0:*1d2*/-1:as/0:*3/`, `0:d/` } → { `0:*1d2*/0:as/0:*3d/` }
{ `0:*1z/-1:as/0:*2/`, `0:z/` } → { `0:*1z/`, `0:as/`, `0:*2z/` }

The rule that attaches to the intermediate result an assignment that makes t_2 true:

{ `0:aq1*/`, `0:bp2*` } → { `0:aq1*/0:bp2*/` }

Compatibility between the both objects is not checked by this rule — the two objects combined may express different assignments (true and false) to the same variable. The following rule starts creating intermediate results that satisfy t_1 and t_2.

{ `0:aq1*/0:bp2*/`, `0:bs/`, `0:b/`, `0:q/` } → { `0:aq1*/0:bp2*/1:bs/0:bq/` }

The next rule assigns true to the variable in the new intermediate result since the previous intermediate result assigned it true and this assignment requires it to be true.

{ `0:*1t2*/0:*3t4*/-1:bs/0:*5/`, `0:t/` } $\rightarrow$ { `0:*1t2*/0:*3t4*/0:bs/0:*5t/` }

What follow are the similar rules for compatible assignments (such as true and don't-care pairs).

{ `0:*1t2*/0:*3d4*/-1:bs/0:*5/`, `0:t/` } $\rightarrow$ { `0:*1t2*/0:*3d4*/0:bs/0:*5t/` }
{ `0:*1d2*/0:*3t4*/-1:bs/0:*5/`, `0:t/` } $\rightarrow$ { `0:*1d2*/0:*3t4*/0:bs/0:*5t/` }
{ `0:*1f2*/0:*3f4*/-1:bs/0:*5/`, `0:f/` } $\rightarrow$ { `0:*1f2*/0:*3f4*/0:bs/0:*5f/` }
{ `0:*1f2*/0:*3d4*/-1:bs/0:*5/`, `0:f/` } $\rightarrow$ { `0:*1f2*/0:*3d4*/0:bs/0:*5f/` }
{ `0:*1d2*/0:*3f4*/-1:bs/0:*5/`, `0:f/` } $\rightarrow$ { `0:*1d2*/0:*3f4*/0:bs/0:*5f/` }
{ `0:*1d2*/0:*3d4*/-1:bs/0:*5/`, `0:d/` } $\rightarrow$ { `0:*1d2*/0:*3d4*/0:bs/0:*5d/` }

Note that there is no rule for incompatible pairs that assign true and false to one variable. The following rule finishes creating second intermediate results.

{ `0:*1z/0:*2z/-1:bs/0:*3/`, `0:z/` } $\rightarrow$ { `0:*1z/`, `0:*2z/`, `0:bs/`, `0:*3z/` }

The next rule makes a pair of the second intermediate result and an assignment that makes t_3 true (again, without checking the compatibility of the variable assignments).

{ `0:bq1*/`, `0:cp2*` } $\rightarrow$ { `0:bq1*/0:cp2*/` }

The rule to start creating final results that satisfy f is

{ `0:bq1*/0:cp2*/`, `0:cs/`, `0:c/`, `0:q/` } $\rightarrow$ { `0:bq1*/0:cp2*/1:cs/0:cq/` }.

The rest of the rules to create final results are obtained by replacing every `bs` with `cs` in the rules for creating second intermediate results.

If an object of the form `0:cq`$x_1x_2x_3x_4$`z` is found in the multiset, the function f is satisfiable and the object represents the assignment to the variables.

When an object is formed from an incompatible pair (such as `0:aqtdddz/` and `0:bpfdddz/` where the assignments to the variable x_1 clash), the object will eventually be transformed to one that does not match any of the given rules; this however does not affect the computability of the system since the necessary materials such as `0:bpfdddz/` or `0:bs/` are supplied by the sources unlimitedly.[6]

5 On implementation

Since this formal model has inherent nondeterminism, there are several possible ways to implement a system described in this model. Excluding implementations using human choices or exhaustive search since those implementations may not cope with a large set of objects with which the system could show interesting behavior, practical implementations may be the following.

- Implementation using random choice. An operation is selected among the possible operations. This seems one of the most effective ways to implement such systems on traditional computers. In this manner we implemented a naive but working interpreter that executes systems described in this model as approximately 450 lines of code in Ruby [6]. This implementation successfully obtained one solution $x'_1x'_3x'_4$ (as the object `0:cqfdffz/`) to the 3SAT problem discussed in Section 4.4 without any problem-specific control.[7]

[6]Since the model is nondeterministic, it is also said that incompatible pairs do not affect computability because choices which do not lead to this kind of block give the final results (if there is at least one solution). The system can also make use of drains to remove such blocked objects from the multiset (such drains can be defined using patterns), or rules that decomposes such objects can be added to the system.

[7]The running time was approximately 46.5 CPU hours on a 1.8GHz Pentium 4 PC with 1GB memory. The size of used memory was 8584KB, and the number of objects in the multiset when the solution was found was 1805.

- Implementation using physical computation mechanisms. If one can find a physical mechanism that behaves in a way similar to the system being described, the execution of the system can be carried out by the physical mechanism. Or other way round, by describing behavior of a physical system in this model, problem solving computation encoded in the description can be carried out by the physical system. Promising options include DNA computing [1].

6 Discussions

Element-conserving dynamic systems could also be modelled using rewriting systems such as generative grammars, term-rewriting systems, and their variants such as Lindenmayer's systems [5]. However, most rewriting systems have a property with which a term can be rewritten to another term comprising arbitrary elements regardless of the elements included in the original term. This property would prevent those systems to be used to describe element-conserving dynamic systems naturally. By contrast, the present model conserves elements. This property of the model can be viewed as constraints on rewriting systems (i.e, this model can be regarded as a restricted rewriting system), but due to this property, the model will naturally captures the characteristics of systems composed of physical material. Moreover, it makes descriptions intuitive (as might be seen in the example descriptions in Section 4), and it gives rise to the possibility of implementation by physical systems.

There are other rewriting models that conserve elements such as Artificial Chemistry [2] and the string-based artificial chemistry [7]. One of the main differences between these and the presented model is that some of the elements in these models are functional, i.e., an element itself has a specific meaning. This characteristic will be useful to evolve self-contained artificial life. In contrast, an element has no function in itself in this model. This makes the model simple and provides the easiness of description, possibly, with the larger feasibility of physical implementation.

In most of other studies of artificial life, systems of creatures are usually modelled in one of the following two ways: (1) objects of a relatively small number of types are controlled by simple interactions with a small number of other objects, or (2) each object is intelligent and controls itself. The former approach has been revealing interesting properties of complex systems composed of simple elements. Specifically, studies based on cellular automata [10] have been showing effectiveness of this approach [3]. The latter approach is also widely taken (although the levels of intelligence each object has differ from system to system) and studies with this approach have been exploring other aspects of complex systems such as the evolution of creatures [4]. Meanwhile, there seem to exist systems of biological activity that can be modelled as element-conserving dynamic systems (like the example given in Section 4.3), which cannot naturally be modelled as a system with either of the two approaches. The presented formal model, together with other systems like Artificial Chemistry and the string-based artificial chemistry, may effectively capture the properties of such systems.

Element-conserving dynamic systems are also closely related to molecular computation by self-assembly. Several formal models have been presented for computation using self-assembly [8, 11]. Those studies have mainly been focusing only on assembling ("composing" in this paper's terminology) molecules, while there might be possibility to use the decomposition of molecules as the computation process. From this viewpoint, element-conserving dynamic systems and molecular computation have a lot in common and the presented formal model might serve as a good tool in this research area.

From the aspect of programming, the simple model described in Section 3 has interesting properties. One of them is indirect imperativeness. Descriptions are given in a similar way to that of term-rewriting systems, in which procedures are not expressed directly and often obscure. As we saw in the examples in Section 4, however, descriptions for a procedural process (such as copying a chromosome by an enzyme) can be given in a relatively straightforward manner, and are not too knotty to comprehend. This might

be due to the property of the model that conserves all the elements through recombination and also the fact that we live in the material world.

7 Concluding remarks

Due to the informal definition, some assumptions of the presented formal model are still implicit. For example, an empty line is allowed both in an object and in a pattern, although the definition might appear to rule that all the elements in an object (or the elementary patterns in a pattern) must be connected. Giving a formal definition of the model is required in order to clarify those assumptions, and it will elucidate the properties of the model as well.

The model assumes the simple structure of objects. Different structures of objects will give different classes of systems, possibly with different computational and/or expressive power. Further study is also necessary from this perspective.

References

[1] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[2] Pietro Sperini di Fenizio and Wolfgang Banzhaf. Stability of metabolic and balanced organisations. In J. Kelemen and P. Sosík, editors, *Advances in Artificial Life: 6th European Conference*, volume 2159 of *LNCS*, pages 196–205, Berlin, 2001. Springer.

[3] Christopher G. Langton. Studying artificial life with cellular automata. *Physica D*, 22:120–149, 1986.

[4] Richard E. Lenski, Charles Ofria, Robert T. Pennock, and Christoph Adami. The evolutionary origin of complex features. *Nature*, 423:139–144, May 2003.

[5] Aristid Lindenmayer. Mathematical models for cellular interactions in development. *Journal of Theoretical Biology*, 18:280–315, 1968.

[6] Yukihiro Matsumoto. *Ruby In A Nutshell*. O'Reilly & Associates, 2001.

[7] Naoaki Ono and Hideaki Suzuki. String-based artificial chemistry that allows maintenance of different types of self-replicators. In *Proceedings of the Fifth International Conference on Humans and Computers (HC-2002)*, pages 173–178, 2002.

[8] Paul W. K. Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the 32nd annual ACM symposium on theory of computing (STOC'00)*, pages 459–468, 2000.

[9] Alan M. Turing. On computable numbers, with an application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936–7.

[10] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana and London, 1966.

[11] Takashi Yokomori. YAC: yet another computation model of self-assembly. In *Proceedings of the 5th DIMACS Workshop on DNA Based Computers*, pages 153–167, 1999.