CDMTCS
Research
Report
Series

Pre-Proceedings of The Workshop on Multiset Processing (WMP-CdeA 2000)

C. S. Calude and M. J. Dinneen (editors)
University of Auckland, New Zealand

Gh. Păun (editors)
Institute of Mathematics of the Romanian Academy, Bucureşti, Romania

CDMTCS-140
August 2000

Centre for Discrete Mathematics and
Theoretical Computer Science

Preface

The **Workshop on Multiset Processing (WMP-CdeA 2000)**, held in Curtea de Argeş, Romania, from 21 to 25 of August, 2000, has the ambitious goal of being the first one in a series devoted to explicitly and coherently developing the **FMT**, the "Formal Multiset Theory", following the experience and the model of **FLT**, the Formal Language Theory.

It starts from two observations: (1) multisets appear "everywhere" (this is also proved by a series of papers in the present volume), and (2) Membrane Computing is a sort of *distributed* multiset rewriting framework, without having equally well developed the non-distributed multiset rewriting (whatever "rewriting" means when dealing with multisets). This is in contrast with what happened in formal language theory, where the grammar system branch has appeared many years after extensively dealing with single grammars and single automata.

As an immediate scope, the workshop is intended to gather together people interested in multiset processing (from a mathematical or a computer science point of view) and in membrane computing (P systems), grounding the development of the latter on the "theory" of the former (although this theory looks rather scattered in this moment, spread in many papers, without any systematic/monographic presentation).

Taking seriously the etymology, the workshop will mean not only presentations, but also discussions, exchange of ideas, problems and solutions, joint work, collaboration. It is quite probable that during this process the present papers will be improved, changed, developed. We advise the reader to take this volume as provisory (as pre-proceedings), mainly meant to be a support for the work during the meeting.

The workshop was organized by the Romanian Academy (by its Institute of Mathematics, Bucharest), the Politechnical University of Madrid (by the Artificial Intelligence Department), the Auckland University, New Zealand (by the Centre for Discrete Mathematics and Theoretical Computer Science), and by "Vlaicu-Vodă" High School of Curtea de Argeş, with the Organizing Committee consisting of Cristian Calude (Auckland), Costel Gheorghe (Curtea de Argeş), Alfonso Rodriguez Paton (Madrid), Gheorghe Păun (Bucharest, chair).

Many thanks are due to all these institutions for their consistent help.

We also thank to the contributors to this volume, as well as to the participants to the workshop.

C. S. Calude
M. J. Dinneen
Gh. Păun

Table of Contents

Arithmetic with membranes[1]

by

Adrian ATANASIU
Faculty of Mathematics, Bucharest University
Str. Academiei 14, sector 1
70109 Bucharest, Romania
E-mail: `aadrian@pcnet.ro`

Abstract: P - systems are computing models, where certain objects can evolve in parallel into an hierarchical membrane structure. Recent results show that this model is a promising framework for solving NP - complete problems in polynomial time.

The present paper considers the possibility to perform operations with integer numbers in a P - system. All four arithmetical operations are implemented in a way which seems to have a lower complexity than when implementing them in usual Computer Architecture.

1 Introduction

For the elements of formal languages we shall use definitions and notations in [6]; for basic notions, notations and results about P - systems [2],[3],[4],[5] can be consulted. In this paper we shall use a variant of P system with Active Membranes, very closed to that defined in [5].

A P *system with active membranes* is a construct

$$\Pi = (V, T, H, \mu, w_1, \ldots, w_m, R)$$

where:

1. $m \geq 1$;

2. V is an alphabet (the total alphabet of the system); its elements are called *objects*;

3. $T \subseteq V$ (the terminal alphabet);

[1]Supported by Spanish Secretaria de Estado de Educacion, Universidades, Investigacion y Desarrollo, project SAB1999-0025

4. H is a finite set of labels for membranes;

5. μ is a *membrane structure*, consisting in m membranes, labeled (not necessarily in a one-to-one manner) with elements of H; there is a (unique) membrane s called *skin*; all the other membranes are inside of the skin.

6. $w_1, w_2, \ldots, w_n$ are strings over V, describing the multisets of objects placed in the m regions of μ;

7. R is a finite set of *development rules*, of the following forms:

   (a) $[_h u \longrightarrow v]_h^\alpha$, for $h \in H,\ u, v \in V^*, u \neq \lambda,\ \alpha \in \{+, -, 0\}$;

   (b) $u[_h]_h^{\alpha_1} \longrightarrow [_h v]_h^{\alpha_2}$, where $u, v \in V^+,\ u \neq \lambda,\ h \in H,\ \alpha_1, \alpha_2 \in \{+, -, 0\}$;

   (c) $[_h u]_h^{\alpha_1} \longrightarrow v[_h]_h^{\alpha_2}$, where $u, v \in V^+,\ u \neq \lambda,\ h \in H,\ \alpha_1, \alpha_2 \in \{+, -, 0\}$;

   (d) $[_h u]_h^\alpha \longrightarrow v$, where $u, v \in V^+,\ u \neq \lambda,\ h \in H,\ \alpha \in \{+, -, 0\},\ h \neq s$;

   (e) $[_h u]_h^\alpha \longrightarrow [_h v_1]_h^{\alpha_1} [_h v_2]_h^{\alpha_2}$, where $u, v_1, v_2 \in V^*,\ u \neq \lambda,\ h \in H,\ \alpha, \alpha_1, \alpha_2 \in \{+, -, 0\},\ h \neq s$;

   (f) $[_{h_0} [_{h_1} u]_{h_1}^+ [_{h_1} v]_{h_1}^-]_{h_0}^\alpha \longrightarrow [_{h_0} [_{h_1} u]_{h_1}^{\beta_1}]_{h_0}^\beta [_{h_0} [_{h_1} v]_{h_1}^{\beta_2}]_{h_0}^\beta$, where $\alpha, \beta, \beta_1, \beta_2 \in \{+, -, 0\}$,
   $h \neq s$.

In [5], rules $(a) - (e)$ are defined only for $u, v \in V,\ u \neq \lambda$ (λ is the empty word); we shell use here a general variant, defined in [3]; this can be reduced to that in [5], but some problems of synchronization can arise.

The rules (e) and (f) have a reduced form here (see also [1]); in a rule of type (e) the membrane h can contain other membranes; also, rules of type (f) are used in [1] with $u = v = \lambda,\ p = 2$.

The rules $(a) - (d)$ are applied in parallel: any objects which can evolve, should evolve. If a membrane with label h is divided by a rule of type (e), which involves an object a, then all other objects and membranes situated in the membrane h which are not changed by other rules, are introduced in each of resulting membranes h. Similarly when using a rule of type (f): the whole contains of the membranes h_0, h_1 are reproduced unchanged in their copies, providing that no rule is applied to their objects.

When applying a rule of type (e) or (f) to a membrane, if there are objects in this membrane which evolve by a rule of type (a), then in the new copies of the membrane the results of evolution are introduced. The rules are applied *bottom - up*, in one step, but first the rules of the innermost region and then level by level until the region of the skin.

When applying a rule of type (b) or (f) it is possible to arise several possibilities. In this case any variant will be accepted.

At one step, a membrane h uses only one rule of types $(b)-(f)$. The skin can never divide. During a computation, objects can leave the skin (by means of rules of type (c)) The terminal objects which leave the skin are collected in the order of their expelling of the system; when several terminal symbols leave the system at the same time, then any ordering of them is accepted.

2 Arithmetical P - systems

Let us consider a basis $q \geq 2$ and

$$x = \overline{a_1 a_2 \ldots a_k} = a_1 \cdot q^{k-1} + a_2 \cdot q^{k-2} + \ldots + a_k,\ a_i \in \{0, 1, \ldots, q-1\},\ k \geq 1$$

be an integer in basis q.

A P-system for x (called here *Arithmetical P - System - APS* for short) can be defined in a natural way as

$$\Pi = (V, T, H, \mu, w_1, w_2, \ldots, w_{n_0}, R),$$

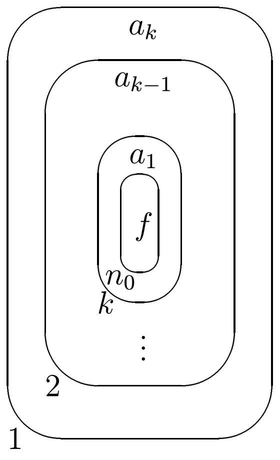
where:

the integer n_0 is a constant fixed by the system (in Computer Architecture structures, $n_0 = 8, 16, 32, 64$ or 128); the examples of this paper uses – without loss of the generality – the value $n_0 = k+1$; $T = \{0, 1, \ldots, q-1\}$, $V \setminus T = \{f\}$, $H = \{1, 2, \ldots, n_0\}$, $\mu = [_1[_2 \ldots [_{n_0}]_{n_0} \ldots]_2]_1$, $w_i = a_{k+1-i}$ $(1 \leq i \leq k)$, $w_i = f$ $(k+1 \leq i \leq n_0)$, R is a set of rules, unspecified in this stage.

Initial, all membranes have a neutral polarity.

Graphically, an APS is represented in Figure 1

Figure 1: The structure of an APS



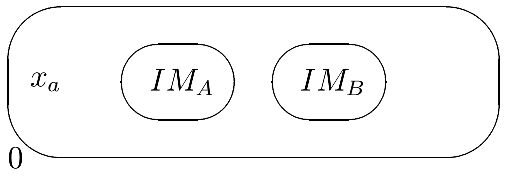
So, in an APS each membrane contains only one object: a digit (terminal object) or f (special nonterminal object); a digit from the membrane i is more significant that all digits situated in the membranes j with $j < i$ and less significant that all digits situated in the membranes j with $j > i$. A f - membrane is the most inner membrane or contains only f - membranes.

We consider here that every APS contains at least one f - membrane.

Because an APS will be placed in other P - systems, the outer membrane 1 of an APS will be not considered the skin.

3 The addition of two APS

In this section we consider that the skin contains two APS. A special object x_a will be the catalyst of operation: the addition of these APS will start in the moment when x_a is placed (somehow) in the skin.



Also, in whole this paper we shall consider the binary case ($q = 2$). The generalization to an arbitrary q is easy to be accomplished.

3.1 Addition with listing

The simplest case we present is the addition of two integers, when the sum is obtained outside the skin. In this situation, no APS remains in the skin (denoted here by membrane s).

Let $a = \overline{a_1 a_2 \dots a_k}$, $b = \overline{b_1 b_2 \dots b_r}$ be two binary integers. We construct a P - system $\Pi = (V, T, H, \mu, w_s, w_1, w_2, \dots, w_{n_0}, R)$,
where: $T = \{0, 1\}$, $V \setminus T = \{x_a, x, y, a, b, f\}$, $H = \{s, 1, 2, \dots, n_0\}$,

$$\mu = [_s[_1[_2 \cdots [_{n_0}]_{n_0} \cdots]_2]_1[_1[_2 \cdots [_n]_n \cdots]_2]_1]_s$$

(polarity is ignored in this construction), $w_s = x_a$, w_i ($1 \le i \le n_0$) defined accordingly with the definition of APS. The set R of rules is defined as follows:

1. $[_s x_a \longrightarrow 0xx]_s$: in skin are introduced 0 (the carry digit) and x - the object which will penetrate membranes. This will be always the first rule applied.

2. $x[_i\ ]_i \longrightarrow [_i y]_i$ ($1 \le i \le n_0$): the object y is introduced by x in the membrane i. This action is performed n_0-times, in parallel for both APS.

3. $[_i y]_i \longrightarrow x$ ($1 \le i \le n_0$): the membrane i is dissolved. This rule acts in tandem with (2).

4. $[_s 000 \longrightarrow 0a,\ 001 \longrightarrow 0b,\ 011 \longrightarrow 1a,\ 111 \longrightarrow 1b]_s$: after the dissolving of the membrane i, two new digits appear in the skin; they react with the carry digit and we obtain a pair: a binary digit – the new carry, and a codification of the sum between these digits (a for 0, b for 1).

5. $[_s 00f \longrightarrow 0a,\ 01f \longrightarrow 0b\ 11f \longrightarrow 1a]_s$: one of the two numbers has finished its digits and offers f; then the sum is accomplished by the carry digit and the digit of the second number.

6. $[_s 0ff \longrightarrow \lambda,\ 1ff \longrightarrow b]_s$: the last membranes were dissolved and two f are free in the skin. The carry digit becomes the most significant digit of the sum; of course, only 1 is kept (usually, a 0 as most significant digit is ignored). Any object f which appears later in the skin will be ignored.

7. $[_s a]_s \longrightarrow 0[_s\ ]_s,\ [_s b]_s \longrightarrow 1[_s\ ]_s$: the sum of two digits is decodified and transported (listed) outside the skin. This operation is synchronized with (2); so the skin will contain in every moment the codification of at most one digit.

Attention: the number obtained as result is in reverse order ! If $c_1c_2c_3\ldots$ is the sequence obtained (c_i after dissolving of membranes labeled with i), the sum of the initial two numbers is

$$c_1 \cdot 2^0 + c_2 \cdot 2^1 + c_3 \cdot 2^2 + \ldots$$

Example 1 *Let us consider the binary numbers* 110 *and* 1011*. We consider (for simplicity)* $n_0 = 5$*; then the initial configuration is*

$$[_s x_a [_1 0 [_2 1 [_3 1 [_4 f [_5 f]_5]_4]_3]_2]_1 [_1 1 [_2 1 [_3 0 [_4 1 [_5 f]_5]_4]_3]_2]_1]_s .$$

This configuration will be transformed step-by-step as follows:

$[_s 0xx [_1 0 [_2 1 [_3 1 [_4 f [_5 f]_5]_4]_3]_2]_1 [_1 1 [_2 1 [_3 0 [_4 1 [_5 f]_5]_4]_3]_2]_1]_s$
$[_s 0 [_1 y0 [_2 1 [_3 1 [_4 f [_5 f]_5]_4]_3]_2]_1 [_1 y1 [_2 1 [_3 0 [_4 1 [_5 f]_5]_4]_3]_2]_1]_s$
$[_s 001xx [_2 1 [_3 1 [_4 f [_5 f]_5]_4]_3]_2 [_2 1 [_3 0 [_4 1 [_5 f]_5]_4]_3]_2]_s$
$[_s 0b [_2 y1 [_3 1 [_4 f [_5 f]_5]_4]_3]_2 [_2 y1 [_3 0 [_4 1 [_5 f]_5]_4]_3]_2]_s$
$1 [_s 011xx [_3 1 [_4 f [_5 f]_5]_4]_3 [_3 0 [_4 1 [_5 f]_5]_4]_3]_s$
$1 [_s 1a [_3 y1 [_4 f [_5 f]_5]_4]_3 [_3 y0 [_4 1 [_5 f]_5]_4]_3]_s$
$10 [_s 011xx [_4 f [_5 f]_5]_4 [_4 1 [_5 f]_5]_4]_s$
$10 [_s 1a [_4 yf [_5 f]_5]_4 [_4 y1 [_5 f]_5]_4]_s$
$100 [_s 11fxx [_5 f]_5 [_5 f]_5]_s$
$100 [_s 1a [_5 yf]_5 [_5 yf]_5]_s$
$1000 [_s 1ffxx]_s$
$1000 [_s bxx]_s$
$10001 [_s xx]_s$

So, the result is $10001 = 110 + 1011$.

It is easy to generalize this construction for any basis q: the rules (1), (2), (3) remains unchanged. The rules from (4) are modified in $ijk \longrightarrow px_p$ where $p = (i + j + k)\ mod\ q$ and $x_p \in V \setminus T$ are new special objects.

The sets (5), (6) of rules are modified similarly. The rules from (7) are now

$$[_s x_p]_s \longrightarrow p[_s\ ]_s,\ p \in \{0, 1 \ldots, q-1\}.$$

The complexity of addition with listing is constant: $\mathcal{O}(n_0)$, because n_0 is a constant beforehand fixed and $2n_0 + 3$ steps are necessary to realise the sum between two APS.

This evaluation can be optimised if we work with a variable number n ($n \leq n_0$) of membranes, but in this case the definition of rules becomes more complex.

3.2 Addition without listing

In the most cases, we need to keep the result into a membrane, in order to use it later on. That's why another construction of the addition of two APS will be realised.

Let us establish the general characteristics of a calculus with membranes we define later on:

- Any membrane from APS contains only one digit or f. The most inner membrane contains f.
- An APS contains at least one digit.
- Both terms of an arithmetical operation have the same number of membranes (n_0 is a constant beforehand fixed).
- The result of the operation can be an APS without f in the most inner membrane, but the number of digits is always at most n_0 (overlaps are not considered).
- All numbers codified in APS's are nonnegative and unsigned.

We can imagine that in the skin there is a membrane – always denoted by 0 – where a single APS – denoted by B – is initially placed (the result of the operation will remain in this APS).

We are interested only what will happen in that peculiar membrane, not in the skin (which can be – for example – a dispatcher for other computing membranes 0); that's why in the following we shall ignore the skin s.

A meta-command **ADD** will introduce in the membrane 0 another APS – denoted by A – with its first membrane polarised $+$, and an object x_a which starts the addition.

After the addition is accomplished, in the membrane 0 remains only the *APS B*, which contains the result. Now, another meta-command concerning the membrane 0 can be produced by the skin.

A meta-command **OUT** will list outside the membrane 0 the digits from the *APS B*.

In order to add two positive integers without listing the result, we shall define a little more complicated P - system. Because the skin s is neutral operational and electrical, in the following s will be sistematic ignored in notations.

Consider the binary integers

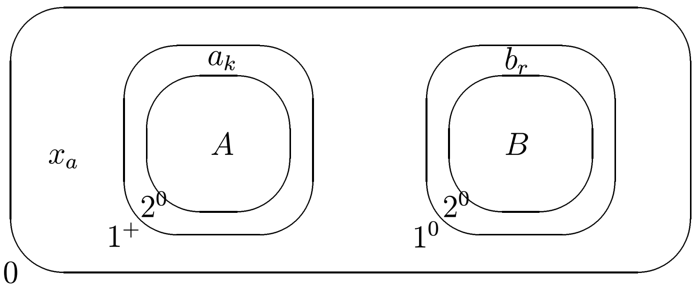
$$a = a_1 2^{k-1} + a_2 2^{k-2} + \ldots + a_k, \qquad b = b_1 2^{r-1} + b_2 2^{r-2} + \ldots + b_r.$$

The P - system will be composed by

$T = \{0, 1\}$, $V \setminus T = \{x_a, x_0, x, y, z, u, v, f, a, b, c, d\}$, $H = \{0, 1, 2, \ldots, n_0\}$,

$$\mu = [_0 x_a [_1 a_k [_2 a_{k-1} \ldots [_{n_0} f]^0_{n_0} \ldots]^0_2]^+_1 [_1 b_r [_2 b_{r-1} \ldots [_{n_0} f]^0_{n_0} \ldots]^0_2]^0_1]^0_0$$

Figure 2: The structure of a P - system for a binary operation



and rules

1. $[_0 x_a \longrightarrow x_0 x_0]_0$, $x_0 [_1]^+_1 \longrightarrow [_1 y]^+_1$, $\quad x_0 [_1]^0_1 \longrightarrow [_1 0]^+_1$.

   At the first step, x_a introduces (via x_0) the objects y in A and 0 in B (which will change its polarity); 0 is the first (virtual) carry digit.

2. $[_i y]^+_i \longrightarrow x$ $(1 \leq i < n_0)$, $\quad [_{n_0} y]^+_{n_0} \longrightarrow z$.

   y dissolves membranes polarized $+$; for the most inner membrane, an object z appears inside the membrane 0; otherwise, the object is x.

3. $x [_i]^0_i \longrightarrow [_i y]^0_i$, $\quad [_i y]^0_i \longrightarrow [_i y]^+_i$ $(1 < i \leq n_0)$.

   x introduces an object y in the neutral membrane i; at the next step, this membrane is polarized $+$ (for synchronization).

4. $p[_i]^\alpha_i \longrightarrow [_i p]^\alpha_i$, $p \in \{0, 1\}$, $\alpha \in \{+, -\}$ $(1 \leq i \leq n_0)$.

   Any digit penetrates through all non-neutral membranes.

5. $$\begin{bmatrix} 000 \longrightarrow ac \\ 001 \longrightarrow bc \\ 011 \longrightarrow ad \\ 111 \longrightarrow bc \end{bmatrix}_i^+ , \quad \begin{bmatrix} 000 \longrightarrow 0 \\ 001 \longrightarrow 1 \\ 011 \longrightarrow 0v \end{bmatrix}_i^0 \quad (1 \le i \le n_0).$$

   The addition $a_i + b_i + c$ is accomplished (c is the carry digit) in the membrane i, polarized $+$. If A has finished its digits earlier and polarization in B is still neutral, then the second variant is used.

6. $$\begin{bmatrix} 00f \longrightarrow ac \\ 01f \longrightarrow bc \\ 11f \longrightarrow ad \end{bmatrix}_i^+ , \quad \begin{bmatrix} 00f \longrightarrow f \\ 01f \longrightarrow 1 \end{bmatrix}_i^0 \quad (1 \le i \le n_0).$$

   B has fewer digits than A, so $b_i = f$ (the sum contains only a_i and the carry digit).

   If both APS have finished their digits but the carry digit has affected the next unpolarized membrane, the second variant is used.

7. $[_i a]_i^+ \longrightarrow [_i a]_i^-$, $\quad [_i b]_i^+ \longrightarrow [_i b]_i^-$, $\quad c[_i]_i^0 \longrightarrow [_i 0]_i^+$, $\quad d[_i]_i^0 \longrightarrow [_i 1]_i^+$ $(1 \le i \le n_0)$.

   The objects a and b change polarity of the membrane i (here the calculation is over); c and d rebuild the carry digit of the next membrane, which is ready for computation of the sum (its polarization becomes $+$).

8. $$\begin{bmatrix} zf \longrightarrow u \\ ff \longrightarrow f \end{bmatrix}_0^0 , \quad \begin{bmatrix} za \longrightarrow 0u \\ zb \longrightarrow 1u \end{bmatrix}_i^0 ,$$

   $u[_i]_i^- \longrightarrow [_i z]_i^0$, $\quad u[_i]_i^+ \longrightarrow [_i 0]_i^0$, $\quad v[_i]_i^0 \longrightarrow [_i 01]_i^0$ $(1 \le i \le n_0)$.

   Final rules: A is dissolved completely, the objects z and u rebuild the neutral polarization of B; v solves the situation when the most inner membrane in B receives a nonzero carry digit.

Example 2 *Let us compute* $11 + 10$. *The sequential transformations of the P - system are the following:*

$$[_0 x_a [_1 1 [_2 1 [_3 f]_3^0]_2^0]_1^+ [_1 0 [_2 1 [_3 f]_3^0]_2^0]_1^0]_0^0$$

$$[_0 x_0 x_0 [_1 1 [_2 1 [_3 f]_3^0]_2^0]_1^+ [_1 0 [_2 1 [_3 f]_3^0]_2^0]_1^0]_0^0 \qquad [_0 [_1 1y [_2 1 [_3 f]_3^0]_2^0]_1^+ [_1 00 [_2 1 [_3 f]_3^0]_2^0]_1^+]_0^0$$

$$[_0 1x [_2 1 [_3 f]_3^0]_2^0 [_1 00 [_2 1 [_3 f]_3^0]_2^0]_1^+]_0^0 \qquad [_0 [_2 1y [_3 f]_3^0]_2^0 [_1 001 [_2 1 [_3 f]_3^0]_2^0]_1^+]_0^0$$

$$[_0 [_2 1y [_3 f]_3^0]_2^+ [_1 bc [_2 1 [_3 f]_3^0]_2^0]_1^+]_0^0 \qquad [_0 1x [_3 f]_3^0 [_1 b [_2 01 [_3 f]_3^0]_2^+]_1^-]_0^0$$

$$[_0 [_3 yf]_3^0 [_1 1b [_2 01 [_3 f]_3^0]_2^+]_1^-]_0^0 \qquad [_0 [_3 yf]_3^+ [_1 b [_2 011 [_3 f]_3^0]_2^+]_1^-]_0^0$$

$$[_0 zf [_1 b [_2 ad [_3 f]_3^0]_2^+]_1^-]_0^0 \qquad [_0 u [_1 b [_2 a [_3 1f]_3^+]_2^-]_1^-]_0^0$$

$$[_0 [_1 zb [_2 a [_3 1f]_3^+]_2^-]_1^0]_0^0 \qquad [_0 [_1 1u [_2 a [_3 1f]_3^+]_2^-]_1^0]_0^0$$

$$[_0 [_1 1 [_2 za [_3 1f]_3^+]_2^0]_1^0]_0^0 \qquad [_0 [_1 1 [_2 0u [_3 1f]_3^+]_2^0]_1^0]_0^0$$

$$[_0 [_1 1 [_2 0 [_3 01f]_3^0]_2^0]_1^0]_0^0 \qquad [_0 [_1 1 [_2 0 [_3 1]_3^0]_2^0]_1^0]_0^0$$

Indeed, $11 + 10 = 101$.

The complexity of the addition is obviously constant – $\mathcal{O}(n_0)$, because all APS have the same depth n_0.

3.3 The Incrementation

The incrementation (addition with 1) is an operation often used in programming languages; thus, it has a separate implementation and an increased execution speed.

We can realise this operation in an easier manner, which will justify its utilisation later on.

Let A be an APS and $\{p,\ +\}$ be two new objects; we shall consider – as usually – only the binary case.

The rules which will be introduced in every membrane of A are:

$$p[_i]_i \longrightarrow [_i+]_i \quad (1 \le i \le n_0),$$

$$\begin{bmatrix} +0 \longrightarrow 1 \\ +1 \longrightarrow 0p \\ +f \longrightarrow 1 \end{bmatrix}_i \qquad (1 \le i \le n_0).$$

So, p starts the incrementation and will be the carry digit (if the actual element of the current membrane is 1). $+$ will add an unit and – in dependence on the other digits – stops the operation or generates a carry digit for the next inner membrane.

4 The Subtraction

4.1 The Decrementation

The decrementation (subtract with 1), can be defined as a special operation with increased speed (like the incrementation). The main problem is to build the most significant digit, because after we subtract one unit, it is possible to remain 0 on the most significant position; all that 0 should be replaced by f.

Five new objects d, d_u, e, e_u and g are used. The rules are (the polarization is always neutral and therefore was omitted):

$$d[_i]_i \longrightarrow [_id_u]_i, \begin{bmatrix} 0d_u \longrightarrow 1d \\ 1d_u \longrightarrow 0e \end{bmatrix}_i (1 \le i \le n_0).$$

d starts decrementation. By penetrating the membrane i, the object d is transformed in d_u; this new object accomplishes the decrementation. When 1 is transformed in 0, the decrementation is finished and an object e is generated, in order to check if that was the most significant 1 or not.

$$e[_i]_i \longrightarrow [_ie_u]_i, \begin{bmatrix} 0e_u \longrightarrow 0d \\ 1e_u \longrightarrow 1 \\ fe_u \longrightarrow fg \end{bmatrix}_i (1 \le i \le n_0).$$

e penetrates the membrane i and becomes e_u. If e_u matches a digit, it will be deleted (and the decrementation is finished). If it matches an f, then a new object g is generated.

$$[_i g]_i \longrightarrow g[_i]_i \; (2 \le i \le n), \qquad \left[\begin{array}{l} 0g \longrightarrow fg \\ 1g \longrightarrow 1 \end{array}\right]_i \; (1 \le i \le n_0).$$

g comes back in the embedding membranes and changes any 0 in f. The first 1 is the most significant digit and g will be eliminated.

Remark: The rule $[0g \longrightarrow fg]_i$ can be avoided; it was introduced as a supplementary precaution, when – by mistake – some 0's remains as most significant digits.

Example 3 *Let us consider the decrementation* $100 - 1$. *The P-system will go through the following transformations:*

$[_0 d[_1 0[_2 0[_3 1[_4 f]_4]_3]_2]_1]_0$ $\quad$ $[_0[_1 0d_u[_2 0[_3 1[_4 f]_4]_3]_2]_1]_0$
$[_0[_1 1d[_2 0[_3 1[_4 f]_4]_3]_2]_1]_0$ $\quad$ $[_0[_1 1[_2 0d_u[_3 1[_4 f]_4]_3]_2]_1]_0$
$[_0[_1 1[_2 1d[_3 1[_4 f]_4]_3]_2]_1]_0$ $\quad$ $[_0[_1 1[_2 1[_3 1d_u[_4 f]_4]_3]_2]_1]_0$
$[_0[_1 1[_2 1[_3 0e[_4 f]_4]_3]_2]_1]_0$ $\quad$ $[_0[_1 1[_2 1[_3 0[_4 fe_u]_4]_3]_2]_1]_0$
$[_0[_1 1[_2 1[_3 0[_4 fg]_4]_3]_2]_1]_0$ $\quad$ $[_0[_1 1[_2 1[_3 0g[_4 f]_4]_3]_2]_1]_0$
$[_0[_1 1[_2 1[_3 fg[_4 f]_4]_3]_2]_1]_0$ $\quad$ $[_0[_1 1[_2 1g[_3 f[_4 f]_4]_3]_2]_1]_0$
$[_0[_1 1[_2 1[_3 f[_4 f]_4]_3]_2]_1]_0$.
Therefore $100 - 1 = 11$.

4.2 The Subtraction of two *APS*

Having defined the addition of two *APS*, the subtraction will be easy to be constructed.

Let be the unsigned integers $a = \sum_{i=0}^{k-1} a_{k-i}q^i$, $b = \sum_{i=0}^{r-1} b_{r-i}q^i$ contained into *APS* A and B respectively. We make the supposition that $a < b$.

Then $b - a = b - \sum_{i_0}^{k-1} a_{k-i}q^i = b + \sum_{i_0}^{k-1}(q - 1 - a_{k-i})q^i - \sum_{i=0}^{k-1}(q-1)q^i = b + \overline{a} + 1 - q^k$.

Hence, to subtract a from b means to add b with the complement of a and with 1; finally, one unit have to be subtracted in position $k+1$.

The algorithm is:

1. $\overline{a} \longrightarrow a$ (A contains the complement of a);
2. $b + 1 \longrightarrow b$ (the incrementation of B – see section 3.3);
3. $a + b \longrightarrow b$ (B contains the sum $\overline{a} + b + 1$);
4. $b - q^k \longrightarrow b$ (one unit is subtracted in the position $k+1$ of B).

Steps (1) and (2) can be accomplished in parallel; moreover, for (2) and (3) the problem is reduced to the addition of two integers. It remains to solve only steps (1) and (4).

Initial, the P - system is shown in Figure 2, with one starting object x_s placed instead of x_a.

There are necessary $2n_0+5$ new objects: $\{x_s, c', d', e', c_0, \ldots, c_{n_0}, x'_0, \ldots, x'_{n_0}\}$ (the objects used in addition, incrementation and decrementation are not encountered; we suppose there are already there). The rules used are:

1. $[_0 x_s \longrightarrow pc'c_0]^0_0$.

   The first step consists in initialization of the objects which will start the four actions of the subtraction. Object p starts the incrementation of B and c' starts the operation of complementarity (of A).

2. The rules used in complementarity of an APS are defined as follows:

$$\begin{array}{ll} c'[_i]^0_i \longrightarrow [_i d']^0_i & (1 \le i \le n_0); \\ [_i x d' \longrightarrow (1-x)c']^0_i & x = 0, 1,\ (1 \le i \le n_0); \\ [_i f d' \longrightarrow e' f]^0_i & (1 < i \le n_0); \\ [_i e']^0_0 \longrightarrow e'[_i]^0_i & (1 \le i \le n_0). \end{array}$$

   Because only A has initially neutral polarity, c' will penetrate the membrane 1 of A and starts the operation of complementarity.

3. When e' arrives in the membrane 0, the operation of addition of these two membranes (A and B) begins:

$$[_0 e' \longrightarrow x_a]^0_0.$$

4. When the addition is performed (section 3.2), each dissolution of a membrane from A modifies a counter:

$$[_0 x c_i \longrightarrow x c_{i+1}]^0_0\ (0 \le i < n_0).$$

5. The first object f appears in the membrane 0 after dissolving of the membrane $k+1$ from A ($k \ge 1$).

$$[_0 c_k f \longrightarrow h x'_k]^0_0. \qquad (i)$$

   x'_k are new objects which penetrates B until the membrane $k+1$ and will start the decrementation beginning with that position. h neutralizes the other apparitions of f (if $k < n$):

$$[_0 h f \longrightarrow h]^0_0. \qquad (ii)$$

   Finally, the rule $[zf \longrightarrow u]^0_0$ from the set (8) of addition rules will be replaced by

$$[_0 z h f \longrightarrow u]^0_0 \qquad (iii)$$

These three rules acts following priorities $(iii) > (ii) > (i)$ because, if an object can evolve, it should evolve !

6. After the addition is finished, the action of x'_k begins:

$$\begin{array}{ll} x'_k[_i]^0_i \longrightarrow [_i x'_k]^0_i & i = 1, \dots k+1, \ (1 \leq k < n_0); \\ [_{k+1} 1x'_k \longrightarrow 0]_{k+1} & [_{k+1} 0x'_k \longrightarrow 1x'_{k+1}]^0_{k+1}; \\ [_{k+1} x'_k \longrightarrow d_u]^0_{k+1}. & \end{array}$$

   Later on, d_u performs the rules defined in the decrementation (section 4.1).

The complexity of subtraction is still constant – it depends only on the number n_0 of membranes which are in an APS.

5 The Product of two APS

To multiply two integers means to add one of the numbers with itself by a number of times equal with the second number. It is a very simple idea, but – for very large numbers – it becomes difficult to be accomplished in a good time.

Let us consider the classical operation of multiplying of the binary numbers a and b. The position of digits from b is essential here. So, if a digit is 0, then the number a is shifted one position to left (this shift corresponds to a multiplying by 2); if the digit is 1, then the actually number a is kept into a temporary location, then it is also shifted. Finally, all the numbers from the temporary locations are added.

For example:
$1101 \times 110 = 11010 + 110100 = 1001110$.

So, the number of integers which will be added equals the number of digits 1 in the second factor of the product.

The P - system which realize the product has initially the same structure with that of addition (see Figure 2), but here the starting object is x_m.

The nonterminal objects used in this operation are $\{x_m, y_0, y, z, x_0, x_1, a, b, v, v_0, v_1, p_0, p_1, \underline{1}, g, f\}$. The rules are:

1. $[_0 x_m \longrightarrow x_0 y_0]^0_0$.

   The first rule to be applied; the object x_0 starts the splitting of B into several membranes, which have were shifted with the powers of 2; y_0 will command the dissolution of the last membrane split.

2. $y_0[_1]^+_1 \longrightarrow [_1 a]^-_1$, $\quad [_1 a \longrightarrow zb]^-_1$, $\quad b[_i]^0_i \longrightarrow [_i b]^-_i \ (1 < i \leq n_0)$, $\quad [_{n_0} b \longrightarrow \lambda]^-_{n_0}$.

In order to separate A from B, all membranes of A are negative polarized. In its membrane 1, the object z is placed.

3. $[_i z]_i^- \longrightarrow y, \quad y[_i]_i^- \longrightarrow [_i z]_i^- \ (1 \leq i \leq n_0)$.

   The membranes of A are dissolved one by one.

4. $1[_1]_1^0 \longrightarrow [_1\underline{1}]_1^0, \quad 0[_1]_1^0 \longrightarrow [_1 v]_1^0, \quad [_1\underline{1}]_1^0 \longrightarrow [_1 v]_1^0[_1]_1^+$.

   The digit obtained by dissolving one of the membranes from A specifies the behavior of B: an 0 starts one shift (designed by the object v), while an 1 makes one copy of B (positive polarized) and starts also a shift for the initial B (neutral).

5. $[_1 vj \longrightarrow 0v_j]_1^0, \quad v_j[_i]_i^0 \longrightarrow [_i p_j]_i^0, \quad [_i p_j t \longrightarrow jv_t]_i^0, \quad [_i p_j f \longrightarrow j]_i^0 \ j, t = 0, 1, \ (2 \leq i \leq n_0)$.

   The shift of elements from B; in the first membrane (the lowest significant digit) a 0 is placed.

6. $[_0 x_0 f \longrightarrow x_0 g]_0^0, \quad g[_i]_i^0 \longrightarrow [_i h]_i^0, \quad [_i xh]_i^0 \longrightarrow g \ (1 \leq i < n_0), \quad [_{n_0} xh]_{n_0}^0 \longrightarrow x_1 \ (x = 0, 1, f), \quad [_0^0 x_0 f \longrightarrow x_0]_0^0$.

   A has no more digits. Therefore B should be entirely dissolved. The object x_1 deletes all next f, finally remaining only one object in the membrane 0, outside all APS.

7. $x_1 y[_1]_1^+ \longrightarrow [_1 x_1]_1^0, \quad [_1 x_1]_1^0 \longrightarrow x_2[_1]_1^0, \quad [x_0 x_2 \longrightarrow x_a]_0^0$.

   The final rules before the addition of temporary locations; x_a starts the addition (section 3.2). The first rule changes the polarization of an APS arbitrarily chosen (where the final result will be collected).

8. The addition of all APS in the membrane 0 is performed. The finally result represents the product between A and B.

   The selection of two APS and the application of the algorithm from 3.2 is not detailed. A meta-command ADD coordinated by the skin can accomplish this operation.

The complexity of the product is very low: only $\mathcal{O}(log\ p)$ where $p = max\{k, r\}$ (remember, k is the number of digits from the integer a, r is the number of digits from the integer b). The number of digits from B assures how many times the operation of shifting is performed. The duplication (of A) and the negative polarization (of all membranes from B) are accomplished in parallel.

Example 4 *Let us perform the multiplication* 110×101. *We will consider* $n_0 = 6$ *(in order to have enough locations in keeping of the result). The computation will be performed using the following transformations:*

$[_0 x_m [_1 0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_1 1 [_2 0 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+]_0^0$

$[_0 x_0 y_0 [_1 0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_1 1 [_2 0 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+]_0^0$

$[_0 x_0 [_1 0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_1 a 1 [_2 0 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^-]_0^0$

$[_0 x_0 [_1 0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_1 z b 1 [_2 0 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^-]_0^0$

$[_0 x_0 y 1 [_1 0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_2 b 0 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^-]_0^0$

$[_0 x_0 [_1 \underline{1} 0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_2 z 0 [_3 b 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^-]_2^-]_0^0$

$[_0 x_0 y 0 \underbrace{[_1 0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+}_{B_1^+} [_1 v 0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_3 1 [_4 b f [_5 f [_6 f]_6^0]_5^0]_4^-]_3^-]_0^0$

$[_0 x_0 B_1^+ [_1 v 0 v_0 [_2 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_3 z 1 [_4 f [_5 b f [_6 f]_6^0]_5^-]_4^-]_3^-]_0^0$

$[_0 x_0 1 y B_1^+ [_1 0 v_0 [_2 p_0 1 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_4 f [_5 f [_6 b f]_6^-]_5^-]_4^-]_0^0$

$[_0 x_0 B_1^+ [_1 \underline{1} 0 [_2 p_0 v_1 0 [_3 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_4 z f [_5 f [_6 f]_6^-]_5^-]_4^-]_0^0$

$[_0 x_0 f y B_1^+ [_1 0 [_2 v_0 0 [_3 p_1 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+ [_1 v 0 [_2 v_0 0 [_3 p_1 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_5 f [_6 f]_6^-]_5^-]_0^0$

$[_0 x_0 g B_1^+ [_1 0 [_2 0 [_3 p_0 v_1 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+ [_1 0 n_0 [_2 0 [_3 p_0 v_1 1 [_4 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0 [_5 z f [_6 f]_6^-]_5^-]_0^0$

$[_0 x_0 B_1^+ f y [_1 0 [_2 0 [_3 0 v_1 [_4 p_1 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+ [_1 h p_0 0 [_3 0 v_1 [_4 p_1 f [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0 [_6 f]_6^-]_0^0$

$[_0 x_0 B_1^+ [_1 0 [_2 0 [_3 0 [_4 p_1 1 [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+ [_2 h v_0 0 [_3 0 [_4 p_1 1 [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0 [_6 z f]_6^-]_0^0$

$[_0 x_0 B_1^+ g f y [_1 0 [_2 0 [_3 0 [_4 v_1 1 [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+ [_3 0 [_4 v_1 1 [_5 f [_6 f]_6^0]_5^0]_4^0]_3^0]_0^0$

$[_0 x_0 y B_1^+ [_1 0 [_2 0 [_3 0 [_4 1 [_5 p_1 f [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+ [_3 h 0 [_4 1 [_5 f [_6 p_1 f]_6^0]_5^0]_4^0]_3^0]_0^0$

$[_0 x_0 y g B_1^+ \underbrace{[_1 0 [_2 0 [_3 0 [_4 1 [_5 1 [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^+}_{B_2^+} [_4 1 [_5 1 [_6 f]_6^0]_5^0]_4^0]_0^0$

$[_0 x_0 y B_1^+ B_2^+ [_4 h 1 [_5 1 [_6 f]_6^0]_5^0]_4^0]_0^0$

$[_0 x_0 y B_1^+ B_2^+ g [_5 1 [_6 f]_6^0]_5^0]_0^0$

$[_0 x_0 y B_1^+ B_2^+ [_5 h 1 [_6 f]_6^0]_5^0]_0^0$

$[_0 x_0 y g B_1^+ B_2^+ [_6 f]_6^0]_0^0$

$[_0 x_0 y B_1^+ B_2^+ [_6 h f]_6^0]_0^0$

$[_0 x_0 x_1 y B_1^+ B_2^+]_0^0$

$[_0 x_0 B_1^+ [_1 x_1 \ldots]_1^0]_0^0$ *(B_2 was arbitrary selected)*

$[_0 x_0 x_2 B_1^+ B_2^0]_0^0$

$[_0 x_a B_1^+ B_2^0]_0^0.$

B_1 *contains* 110 *and* B_2 *contains* 11000. *After the addition is performed, the final result (collected into* B_2*) is* $[_0 [_1 0 [_2 1 [_3 1 [_4 1 [_5 1 [_6 f]_6^0]_5^0]_4^0]_3^0]_2^0]_1^0]_0^0$,
that means $110 \times 101 = 11110$.

Remark: If a product of a binary integer by 2 is required, only the step (5) from the algorithm is used. This corresponds in *Computer Architecture* to a **shift to right** operation and it has a separate faster implementation (similar to the operations of incrementation and decrementation).

6 The Division

The division of two integers is a little more complicated. Having two APS corresponding to a (nominator) and b (denominator), the algorithm of division will work in three steps:

1. At first, two other APS for quotient (q) and remainder (r) will be generated;

2. By decrementing q and r, new membranes 0 will be constructed, each membrane containing four APS for these integers (a, b, q, r).

3. In parallel, in each membrane 0 one verifies if the equality $a = bq + r$ holds. The membrane where this assertion is true will keep the values q and r (the APS A and B are dissolved). All the other membranes are dissolved.

The P - system will have as new objects $\{x_d, x', x", z, a, b, a', b', a_1, b_1, q, q_1, q_2, r, r_0, r_1, r_2, r_3, \dagger, \triangle\}$. Its initial structure is that from Figure 2, with x_d instead of x_a (remember, the membrane 0 is not the skin; the skin s was not drawn).

The rules are:

1. $[_0 x_d \longrightarrow x'x"]_0^0, \quad x'[_1]_1^0 \longrightarrow [_1 z]_1^0, \quad x"[_1]_1^+ \longrightarrow [_1 z]_1^+.$

   The first rules which will be applied. The two APS are prepared to be split.

2. $[_1 z]_1^0 \longrightarrow [_1 a]_1^+ [_1 q]_1^0, \quad [_1 z]_1^+ \longrightarrow [_1 b]_1^+ [_1 r]_1^0.$

   Two new APS Q – for the quotient and R - for the remainder, are created. All APS contain into their first membrane a stamp (a, b, q, r) to identify which integer is stored. We shall identify this stamp x with the value of integer kept in APS X $(x = a, b, q, r)$.

3. The rules for decrementation of Q and R are introduced. The initial value for q is $a - 1$, and for r is $b - 1$.

   When the decrementation is finished q is replaced by q_1 into Q, r with r_1 into R. Their polarization remains still neutral (A and B are positive polarized).

4. $[_1 q_1 x]_1^0 \longrightarrow [_1 q_1 x]_1^+ [_1 q_2 x]_1^- \; x \in \{0, 1\}, \quad [_0 [_1 q_1]_1^+ [_1 q_2]_1^-]_0^0 \longrightarrow [_0 [_1 q]_1^0 [_0 [_1 q_2]_1^+]_0^0.$

   If Q contains a nonzero integer, then the membrane 0 is split in two other membranes, each of them containing the four APS (A, B, Q, R). The process (the decrementation and the splitting) continues with that membrane 0 which contains Q, neutral polarized.

5. $[_1 q_2 \longrightarrow r_0 q]_1^+, \quad [_1 r_0]_1^+ \longrightarrow r_0 [_1]_1^+, \quad r_0 [_1]_1^0 \longrightarrow [_1 r_1]_1^0, \quad [r_1 r_1 \longrightarrow r_2]_1^0$

   In a membrane 0 with Q polarized $+$, only R may have a neutral polarization. These rules transfer from Q to R properties to decrementing and splitting (copies of A, B and Q are automatically produced in the membrane 0 at each splitting).

6. $[_1 r_2 x]_1^0 \longrightarrow [_1 r_2 x]_1^+ [_1 r_3 x]_1^-, \ x \in \{0,1\}, \quad [_0 [_1 r_2]_1^+ [_1 r_3]_1^-]_0^0 \longrightarrow [_0 [_1 r_1]_1^+]_0^0 [_0 [_1 r_1 r]_1^0]_0^0.$

   One of these two membranes 0 obtained contains four APS positive polarized; this membrane stops its splitting and it is ready to check the relation $a = bq + r$. In the other membrane 0, the APS R is neutral and contains r; therefore, it will be decremented and the string $r_1 r_1$ obtained leads to r_2, so to another (possible) splitting.

7. $[_1 f r_2]_1^0 \longrightarrow [_1 0 r]_1^+.$

   The case $r = 0$; the last remainder generated.

8. $[_1 r_1]_1^+ \longrightarrow x_s x_m [_1]_1^-.$

   All APS are positive polarized. The APS R produces the objects x_s which will start the subtraction $a - r$ (accordingly with 4.2) and x_m will start the product $b \cdot q$ (accordingly with section 5). These results are obtained in A and B respectively.

   After this step, all APS from such a membrane 0 are negative polarized. The rules which accomplishes this restriction are easy to be defined.

9. $$\begin{array}{lll} [_i a]_i^- \longrightarrow a' & & \\ [_i b]_i^- \longrightarrow b' & (1 \le i < n_0) & \\ a_1 [_i]_i^0 \longrightarrow [_i a]_i^- & & \\ b_1 [_i]_i^0 \longrightarrow [_i b]_i^- & (1 < i \le n_0) & \\ [_{n_0} a f]_{n_0}^- \longrightarrow \lambda & & \\ [_{n_0} b f]_{n_0}^- \longrightarrow \lambda & & \end{array} \qquad \left[\begin{array}{l} a'b'ff \longrightarrow a_1 b_1 \\ a'b'00 \longrightarrow a_1 b_1 \\ a'b'11 \longrightarrow a_1 b_1 \\ a'b'01 \longrightarrow \dagger \\ a'b'0f \longrightarrow \dagger \\ a'b'1f \longrightarrow \dagger \end{array} \right]_0^0 .$$

   One verifies if the contents of the APS A and B are equals. These two APS are dissolved. If the answer is YES, in the membrane 0 remains only tho APS: Q and R.

10. $$\begin{array}{ll} \dagger [_1]_1^\pm \longrightarrow [_1 \dagger]_1^0, & [_1 \dagger]_1^0 \longrightarrow \dagger \triangle, \\ \dagger [_i]_i^0 \longrightarrow [_i \dagger]_i^0, & [_i \dagger]_i^0 \longrightarrow \dagger \ (1 < i \le n_0), \\ [_0 \triangle]_0^0 \longrightarrow \lambda, & [_0 \triangle x \longrightarrow \triangle]_0^0 \ x = 0, 1, f, \\ [_1 q_1 f \longrightarrow \dagger]_1^0. & \end{array}$$

    The APS A and B contain different values; then the whole embedding membrane 0 is dissolved.

    The last rule is used when Q contains the value 0 (after the last decrementation). Then the membrane 0 is not able to check if $a = bq + r$ holds, therefore it should be dissolved.

It is interesting to see that the complexity of this algorithm is linear $\mathcal{O}(a+b)$. Indeed, a steps are necessary in generating the 0 membranes with $q = a-1, a-2, \dots, 1$ (the last membrane, for $q = 0$ will be immediate dissolved). The generating of b 0 - membranes for values of $r = b-1, \dots, 0$ is commited in parallel with q, so this does not spend time, unless the last generating – for $q = 1$ – where these b steps are encountered.

The subtraction is constant and the product has a lower complexity. The dissolution of membranes spends also a constant time.

Some optimisations can be made to this algorithm.

For example, when a 0 membrane reaches the correct result, from here an object can be eliberated in the skin, object which "kills" all the other 0 membranes (because all of them will failed later on).

So, the rule $[_0 a'b'ff \longrightarrow a_1b_1]_0^0$ from group (9) is replaced by $[_0 a'b'ff \longrightarrow a_1b_1\$]_0^0$, where \$ is a new object. \$ is eliberated in the skin and "viruses" with $\dagger$ all the other 0 membranes.

This optimization is easy to be constructed, but \$ finally remains in the skin and we have care to protect the other membranes which will be generated in the skin later on (by other meta-commands).

7 Final remarks

We have shown that (and how) an arithmetical calculus can be defined in the P - systems framework. Such a calculus can be the basis for more elaborated applications, maybe also for constructing computer chips or for solving general mathematical problems. Anyway, it seems that complexity of this calculus is lower than that on which actual computers are based.

The whole construction from this paper is purely theoretical, the validation of the discussed ideas should be made in a biochemical framework.

References

[1] Krishna, S.N., Rama, R - *A variant of P - systems with active membranes: Solving NP - complete problems*, Romanian J. of Information Science and technology, 2, 4 (1999), 357-367.

[2] Paun, Gh. - *Computing with membranes*, Journal of Computer and System Science, 2000, and Turku Center for Computer Science - TUCS Report No. 208, Nov. 1998 (www.tucs.fi).

[3] Paun, Gh. - *Computing with membranes. An introduction*, Bull. of the EATCS, 67 (Febr. 1999), 139-152.

[4] Paun, Gh. - *Computing with membranes - A variant: P systems with polarized membranes*, Intern. J. of Foundations of Computer Science, 11, 1 (2000), 167 - 182, and Auckland University CDMTCS Report No. 098, 1999 (www.cs.auckland.ac.nz/CDMTCS).

[5] Paun, Gh - *Computing with membranes (P - systems); Attacking NP - complete problems*, Unconventional Models if Computing (I. Antoniou, C.S. Calude, M.J. Dinnen, eds.), Springer - Verlag, 2000 (in press).

[6] *Handbook of formal languages*, vol. 1, Rozenberg, G, Salomaa, A (Eds), Springer Verlag 1997.

Programming by multiset transformation: a review of the Gamma approach

Jean-Pierre Banâtre*

Extended abstract.

The Gamma paradigm was originally introduced in [1] and has been developed further by a number of researchers [2]. It allows algorithms to be expressed without introducing any superfluous sequentiality which would not be required by the very logic of the algorithms. Essentially, Gamma uses only one data stucture, the multiset, and one control structure, the rewriting through the Gamma operator. The multiset rewritings can be compared with chemical reactions which consume elements of the multiset while producing other elements as the reaction product.

Gamma is a very high level programming language, clearly defined by an operational semantics, and which can be implemented in a rather straightforward way, although in practice, information on the target architecture is welcome to produce an efficient implementation.

Gamma programs consist of applications of conditional rewriting rules to a multiset. These rules have the following form:

$$x_1, \ldots, x_n \to A(x_1, \ldots, x_n) \Leftarrow R(x_1, \ldots, x_n)$$

in which the reaction condition R is a predicate, and the action A is a function operating on a multiset of data elements. An application of this rule consists of finding, if possible, elements $x_1, \ldots, x_n$ of the multiset such as $R(x_1, \ldots, x_n)$ is true and replacing them by the result of the application of $A(x_1, \ldots, x_n)$. The process is repeated until it is not possible to find any new tuple such as $R(x_1, \ldots, x_n)$. At this point of the computation, hopefully, the resulting multiset is the answer.

As an example consider the Gamma version of a program computing the maximum of a set of values.

$$\max : x, y \to y \Leftarrow x \leq y.$$

The same program written in a more traditional language would use an array for an imperative language or a list for a declarative language and would use an iteration to explore the array or a recursive walk through the list. So, the data structure would impose constraints on the order in which the elements would be processed.

*Université de Rennes 1 and Inria, France. Email: Jean-Pierre.Banatre@inria.fr

The essential feature of Gamma programming style is that a data structure is no longer seen as a hierarchy that has to be decomposed by the program in order to access atomic elements. Atomic values are gathered into one single bag and the computation is the result of their individual interactions. A related notion is the "locality principle" in Gamma: individual values may react together and produce new values in a completely independent way. As a consequence, a reaction condition cannot include any global property on the multiset. The locality principle is crucial as it makes it easier to reason about programs and allows an highly parallel interpretation of Gamma programs.

A formal semantics of the language in terms of multiset rewriting has been proposed and discussed in [3]. In this paper, techniques have been proposed in order to prove properties of programs and to derive programs from specifications. Without going into details here, let us mention an interesting property of multisets which is very useful to produce termination proofs for gamma programs. To this purpose, we can use a result from [4] allowing the derivation of a well-founded ordering on multisets from a well-founded ordering on elements of the multiset.

Let $\succ$ be an ordering on V and $\gg$ be the ordering on $Multisets(V)$ defined in the following way:

$$\begin{gathered} M \gg M' \Leftrightarrow \\ \exists X, Y \in Multisets(V). \quad X \neq \emptyset \text{ and} \\ X \subseteq M \text{ and } M' = (M - X) + Y \text{ and } (\forall y \in Y, \quad \exists x \in X. \quad x \succ y). \end{gathered}$$

The ordering $\gg$ on $Multisets(V)$ is well-founded if and only if the ordering $\succ$ on V is well-founded. This result is fortunate because the definition of $\gg$ precisely mimicks the behaviour of Gamma (removing elements from the multiset and inserting new elements). The significance of this result is that it allows us to reduce the proof of termination, which is essentially a global property, to a local condition.

Our presentation will emphasize the very minimal nature of the Gamma formalism as a key factor which makes possible the development of elegant programs in a very rigourous way. Of course, the elegance and power of the multiset data structure is central to the Gamma approach. We will also review some of the work which has been done here and there on the gamma paradigm or on the chemical reaction model (which has been derived from the original Gamma model). Three kinds of contributions will be developed:

1. the relevance of the Gamma model to program development and software engineering,
2. some extensions of the original model, concerning in particular, data structuring facilities and
3. implementation issues.

References

[1] J.-P. Banâtre and D. Le Métayer, Programming by multiset transformation, Communications of the ACM, Vol. 36-1, pp. 98-111, january 1993.

[2] J.-P. Banâtre and D. Le Métayer, Gamma and the chemical reaction model: ten years after, in Coordination programming: mechanisms, models and semantics, Imperial College Press, Andreoli, Hankin and Le Métayer editors, pp. 3-41, 1996.

[3] J.-P. Banâtre and D. Le Métayer, The Gamma model and its discipline of programming, Science of Computer Programming, Vol. 15, pp. 55-77, 1990.

[4] N. Dershowitz, Z. Manna, Proving termination with multiset ordering, Communications of the ACM, Vol. 22-8, pp. 465-476, august 1979.

Data Structures for Implementing Transition P System in Silico[*]

Baranda A., Castellanos J., Molina R.
Dpto. de Inteligencia Artificial
Facultad de Informática - U.P.M.
Campus de Motegancedo, Boadilla del Monte
28660 Madrid - SPAIN
`jcastellanos@fi.upm.es`

Arroyo F., Mingo L.F.
Dpto. de Lenguajes, Proyectos y Sistemas Informáticos
Escuela de Informática - U.P.M.
Carretera de Valencia Km. 7
28031 Madrid - SPAIN
{ `farroyo, lfmingo` }`@eui.upm.es`

Abstract

P Systems introduce a new parallel and distributed computational model. They are based in the membrane structure notion. The constituent structure of a P System is built by some membranes recursively placed inside a special and unique membrane named *skin*. Each membrane defines a region in which objects can be placed. Objects inside membranes are able to evolve, that is, they can be transformed in other objects, can go throw a membrane, or even this evolution can produce a membrane dissolution in which objects are placed. The way in which objects of a P System evolve is by rules execution. Associated to each membrane there are objects and rules, evolution is performed by execution of all the rules inside each membrane of a P System that they can be executed over objects placed in the same membrane. Rules execution is done in a parallel and non-deterministic way. Through P Systems evolution, we get computational devices starting in S0 estate. This initial estate has been obtained putting objects and rules inside each membrane of the P System. Them we let the system goes on until there are no objects able to evolve. At this point, we say that computation has finished and the result is determined by the number of objects situated in a determined membrane of the P System. Because of non-determinism in P Systems, in some occasions, P Systems dont stop their evolution, then we cannot obtain any output from them and it is said that computation is not valid. Where implementing P Systems is actually an open problem. It can be though to be implemented in living being ("in vivo") or in traditional computers either. Implementation in digital computers of P Systems

[*]Thanks to George Păun for his stay at the Technical University of Madrid - Spain.

can be a difficult task, over all in order to obtain high degree of parallelism and non-determinism exhibit by P Systems. However it is an interesting challenge for computer researchers and they have already done some attempts to simulate them -some variant of P Systems- in a digital computer.

This paper explores different data structures to facilitate Transition P Systems implementation in a digital computer. It is structured in a constructive manner to facilitate the comprehension of final representation of Transition P Systems into proposed data structures. Firstly, we present a theoretical presentation of Transition P Systems including the necessary notation to understand their operational mode. Secondly, we study different data structures in which are possible to represent them. Finally, we find out a computational paradigm in order to determine the feasibility of simulating Transition P System by a program running in a digital computer.

Keywords: Membrane structure, P System, Data structure, Natural Computing.

1 Introduction

Within Natural computing area P System is a new computing model based on the way nature organises cellular level in living organisms [1, 2]. Different processes developed at this level can be thought as computations.

Among different variants of P Systems, we have chose transition P Systems as objects of our study to try to translate their structure into data structures what it will permit their simulation in digital computers.

Transition P Systems have two different components: the static one or super-cell, and the dynamic one compounds by evolving rules. Evolving rules define objects evolution in the system and they can eve change the static component by dissolving membranes from super-cell.

Going towards definition of super-cell, we can say that a super-cell is a hierarchical structure of biuniquely labelled membranes. A membrane can contain none, one o more than one membrane. The external one is named *skin* and it can never be dissolved. Membranes without inner membranes are called elemental membranes.

Membranes define regions. We name region to the area enclosed by a membrane and this area is not enclosed by any inner membranes to the first one. Regions contain objects that evolve following evolution rules associated to the membrane. Objects are symbols from an alphabet.

For rules associated to a membrane, we can define a priority relation what define a partial order among rules for each membrane. We can think that rules consume objects from the region they are associated and send objects to their regions or to adjacent regions to their regions.

The transforming process due to evolving rules in transition P Systems is done in parallel in every region of the P System. Moreover, inside each region, at the same step, in an exhaustive manner, every executable rule is executed in the same step, in a non-deterministic way. That is, if there are several possibilities to execute rules, in a determined step, the system are free to execute one of them and every executable rule is executed in parallel at the same step. Some rules associated to a region have the capability of dissolving the membrane; they must to have as consequent the delta symbol.

Transition P System can make computation sending objects to a determined region of the system [2, 3]. This output region is named i_0. We can say that a computation is finished when any rule can be executed in the system.

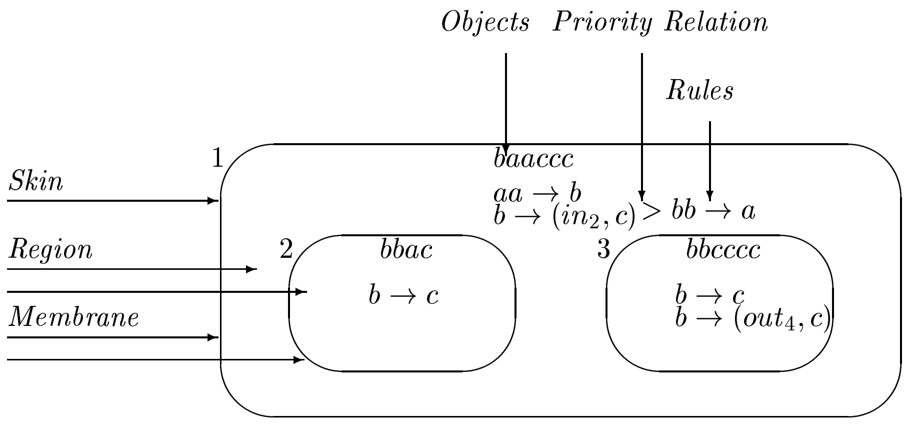


Figure 1: Membrane architecture, including objects, relations and rules.

One of the most important problems in simulating systems is to decide which data structure is the most adequate to representing the system and the needed additional information to simulate the dynamic component of it, in an efficient manner. This paper explores two possibilities one based on array data structure, and the other one based on tree data structure.

2 Theoretical presentation of Transition P Systems

In this, point Transition P Systems are presented in a constructive manner. Firstly we will define the needed basic concepts and finally we will joint adequately in order to define a Transition P System [1, 2, 5].

2.1 Membrane Structure

Now we are going to give some necessary definitions to define and understand membrane structure of Transition P Systems.

Let the language MS be defined recursively as:

(a) $[\,] \in MS$

(b) if $\mu_1 \cdots \mu_n \in MS$ then $[\mu_1 \cdots \mu_n] \in MS$

(c) objects defined by 1 or 2 only belongs to MS

A membrane structure μ is defined as a word belonging to the language MS over an alphabet $\{[,]\}$. Let μ a membrane structure then to each pair of $[,]$ is named membrane. The external one is named *skin*. Every membrane of μ having the form $[\,]$, without word concatenation from MS between both symbols is named elemental membrane.

The degree of a membrane structure μ or number of membranes of μ is denoted by $deg(\mu)$ and it is recurrently defined by:

(a) $deg([\,]) = 1$

(b) $deg([\mu_1 \cdots \mu_n]) = 1 + \sum_1^n deg(\mu_i)$

Let us define now the depth of a membrane structure μ by $dep(\mu)$. It is recursively defined by:

(a) $dep([\,]) = 1$

(b) $dep([\mu_1 \cdots \mu_n]) = 1 + max\{dep(\mu_1), \cdots, dep(\mu_n)\}$

In a membrane structure μ, the number of regions is equal to $deg(\mu)$. A natural way to represent membrane structures is by Venn Diagrams, see figure 2. This kind of representation is very useful to clarify the notion of region defined above as every closed space delimited by membranes. We say two regions are adjacent if and only if there is only one membrane between them. Communication between two regions is possible if and only if regions are adjacent [3, 4].

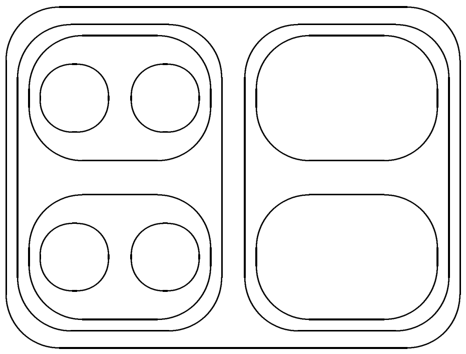


Figure 2: Venn diagram of a membrane architecture.

2.2 Multisets

In this point we will give a compact representation to multisets as word generated by a given alphabet.

Let N the natural number set, and let U an arbitrary set. A multiset over U is a mapping $M : U \to N$. For every $a \in U$, $M(a)$ = multiplicity of a in M. We indicate this fact also in the form $M = \{(a, M(a))/a \in U\}$.

Some more definitions about multiset: Let $M : U \rightarrow N$ a multiset. The support of M is denoted by $supp(M)$ and it is defined by $supp(M) = \{a/a \in U \wedge M(a) > 0\}$. Size of M is denoted by $size(M)$ and it is defined by $size(M) = \sum M(a)$ with a varying in U. A Multiset is empty if and only if $supp(M) = \{\}$ if and only if $size(M) = 0$.

Let $M1$ and $M2$ two multiset over U. Then $M1$ is included in $M2$ iff for every $a \in UM1(a) <= M2(a)$. The union of $M1$ and $M2$ is defined by: $M1 \cup M2 : U \rightarrow N/(M1 \cup M2)(a) = M1(a) + M2(A)$ for every $a \in U$.

Let $M1$ and $M2$ two multiset over U and $M2$ is included in $M1$. Then the multiset $M1 - M2$ is defined by: $M1 - M2 : U \rightarrow N$ given by $(M1 - M2)(a) = M1(a) - M2(a)$ for every $a \in U$.

Let M a multiset of finite support, that is, $card(supp(M) = n \in N$, let $M' = \{(a, M(a))/a \in supp(M)\} = \{(a_1, M(a_1)), \cdots, (a_n, M(a_n))\}$. Then M' -the multiset in which elements of cero multiplicity have been removed- can be represented by any word resulting of permuting $a_1^{M(a_1)} \cdots a_n^{M(a_n)}$. It is necessary noted that empty multiset over U is represented by λ.

An alphabet is a finite nonempty set of abstract symbols. Given an alphabet V we define V^* as the set of all finite words of elements in V, including empty word λ. Length of a word x belonging to V^* is denoted by $|x|$. The number of occurrences of an abstract symbol "a" from the alphabet V in a word x of V^* is denoted by $|x|_a$. A set of words is named language. Every word x from V^* describe a multiset over V denoted by $m(x)$ and defined by: $m(x) = \{(a, |x|a)/a \in V\}$ it is necessary to note that:

(a) $m(\lambda) = \{(a, 0)/a \in V\}$

(b) $supp(m(\lambda)) = \{\ \}$

(c) $size(m(x)) = 0$

$$aabbcc = a^2b^2c^2 = \{(a, 2), (b, s), (c, 2)\} \tag{1}$$

$$a^{n_a}b^{n_b} \cdots z^{n_z} = \{(a, n_a), (b, n_b), \cdots, (z, n_z)\} \tag{2}$$

2.3 Supper-cell concept

A super-cell is defined from definition of concept previously introduced: multiset and membrane structure.

Let U a denumerable set of objects. Let μ a membrane structure with its membranes biuniquely labelled with natural number from 1 to $deg(m)$. Regions associated to membranes in m are labelled automatically too.

A partial order relation $<_\mu$ can be defined over $\{1, \cdots, deg(\mu)\}$ by μ Let $i, j \in \{1, \cdots, deg(\mu)\}$ two labels from different regions of μ. Then $i <_\mu j$ iff region i contains region j directly or i contains a region k which contains j.

Now we can define concept of adjacent in a formal way. Let $i, j \in \{1, \cdots, deg(\mu)\}$ two labels from two regions of μ, we say that i is adjacent to j iff:

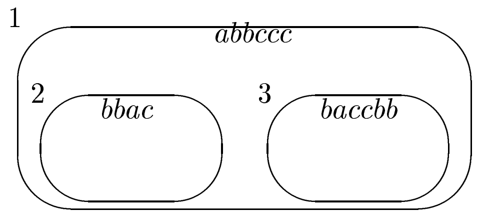


Figure 3: Membrane architecture with statics components.

$$(i <_\mu j \wedge \nexists k \in \{1, \cdots, deg(\mu)\} / i <_\mu k \wedge k <_\mu j) \vee$$
$$(j <_\mu i \wedge \nexists k \in \{1, \cdots, deg(\mu)\} / j <_\mu k \wedge k <_\mu i)$$

A super-cell is obtained associating to each region of a membrane structure μ a multiset over U of objects in such a manner that: for i varying from 1 to $n = deg(\mu)$, labelled region i has associated the multiset $M_i : U \to N$. M_i is said to be the content of region i.

By now we have the static component of a Transition P System (figure 3), this part brings structure and content to the computational device, but it needs dynamic part bringing by evolution rules.

2.4 Evolution rules: Algebraic Definition and their Representation

We here will define algebraically evolution rules from a multiset and from a linguistically point of view.

Let N the natural number set. Let U set of objects. An evolution rule is a terna (u, v, δ), where:

i. u is a multiset over U

ii. v is a multiset over $U \times (\{here, out\}) \cup \{in_j / j \in N\})$

iii. δ is a boolean value

Let $r = (u, v, \delta)$ an evolution rule. Radius of r is denoted by $radius(r) = size(u)$.

If we have into account that we can represent multiset over U as word from an alphabet U, we can redefine rules as follow.

Let N the natural number set. Let V an alphabet whose elements (abstract symbols) are named objects. An evolution rule is a terna (u, v, δ) where:

i. u is a word over V

ii. v is a word over $V \times (\{here, out\}) \cup \{in_j / j \in N\})$

iii. δ is a boolean value

Let $r = (u, v, \delta)$ an evolution rule. Radius of r is denoted by $radius(r) = |u|$. Usually, a rule is represented by:

iv. $u \to v\delta$ if δ

v. $u \to v$ if $\neg\delta$

u is said to be the antecedent and v the consequent rule.

2.5 Transition P System: Algebraic Definition

Now we are going to give a formal definition of a Transition P System [5]. This definition will be based on previous concepts and definitions: Super-cell and evolution rules.

Let N the natural number set. Let U a set (alphabet) whose elements are named objects. A Transition P System over U is a construct

$$PS = (\mu, \omega_1, \cdots, \omega_{deg(\mu)}, (R_1, \rho_1), \cdots, (R_{deg(\mu)}, \rho_{deg(\mu)}), i_0) \tag{3}$$

Where:

i. μ is a membrane structure with biuniquelly labelled membranes with natural number from 1 to $deg(\mu)$.

ii. ω_i, $1 \leq i \leq deg(\mu)$, is a multiset (word) over U associated to region i.

iii. R_i, $1 \leq i \leq deg(\mu)$, is a finite set of evolution rules associated to region i.

iv. $<_i$, $1 \leq i \leq deg(\mu)$, is a partial order over R_i.

v. $1 \leq i_0 \leq deg(\mu)$, is a label to determined output membrane of the system.

First part of the P System $(\mu, \omega_1, \cdots, \omega_{deg(\mu)})$ defines one supe-cell, it is the static part of the system. The second one $((R_1, \rho_1), \cdots, (R_{deg(\mu)}, \rho_{deg(\mu)}), i_0))$ is the dynamic part of the system giving evolving capacity to it.

In this case, we have defined a Transition P System with a membrane output (figure 4), but it is possible to define some different variant without output membrane, or with multiples output membranes either.

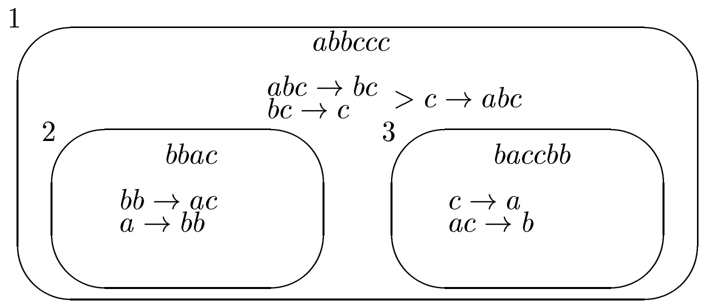


Figure 4: Dynamic components of a membrane architecture.

Now we are ready to understand how a Transition P System works.

2.6 Dynamic Description of P System

Initially, we will give an informal description of Transition P Systems operational mode. Afterwards, we will describe it more formally.

Inside each region defined by a membrane system, evolution rules associated to this region describe possibilities of evolution for objects inside it. We say that an evolution rule is useful, when all regions referred in the consequent rule exist in the system and all of them are adjacent to the region where the rule belongs to. We say that an evolution rule is potentially active in a region when its antecedent is contained in the multiset associated to the region. We say that a rule is active in a region when there are no evolution rules in the region with higher priority. In a region, objects evolve following active rules of the region.

Now, if we have a multiset of objects which can evolve following a multiset of active evolution rules, in parallel, and it is not determined by which of active multiset evolution rules then they will evolve but it is sure that the multiset object will do.

When a multiset u of objects in a region R evolve following an evolution rule $r = (u, v, \delta)$, following effects are produced in the region:

i. object involved on u is removed from region R.

ii. objects referred in v are sending to their destinations.

iii. if δ, then membrane containing R is dissolved. This fact has following efects:

   iii.a. Objects contained in R are associated to Rs father region.

   iii.b. Rules associated to R disappear

   iii.c. Membranes included in R now are included directly in Rs father region.

These effects are happened in parallel for every region of P System. Because of this fact, it can be said that a P System has two parallelism levels; one local or regional and another one global.

The main reason for labelling membranes in a biuniquelly manner, with no dependence of their position into membrane structure, is the necessity of referring membranes without ambiguity although some of them were dissolved.

We will now give some definition that they will be used from now on going.

2.6.1 Maximal set of a given set from a partial order relationship

Let $<$ *a* partial order relationship over $U \times U$. Let S a sub-set from U. We define maximal set of S respect $<$ as:

$$Maximal_{<}(S) = \{x | (x, y) \in S \times S \wedge \neg(x < y)\} \tag{4}$$

2.6.2 Representation of parallel application of evolution rules

In order to represent parallel application of evolution rules we denote application of r_1 rule n_1 times, $\cdots$, r_m rule n_m times as the multiset $\{(r_1, n_1), \cdots, (r_m, n_m)\}$.

2.6.3 Addition of evolution rules

Let $r_1 = (u_1, v_1, \delta_1)$ and $r_1 = (u_2, v_2, \delta_2)$ two evolution rules associated to the same region R. Then evolution rule $r_1 + r_2 = (u_1 \cup u_2, v_1 \cup v_2, \delta_1 \vee \delta_2)$ by definition. This binary inner operation defined over evolution rules set inherits all algebraic properties from multiset union and boolean disjunction.

From the dynamic point of view of P Systems, we can say that are equivalent:

i. Objects from a region R evolved using in parallel the multiset defined by the expression $\{(r_1, n_1), \cdots, (r_m, n_m)\}$.

ii. Objects from a region R evolved using the rule $n_1 r_1 + \cdots + n_m r_m$

From a computational point of view, application of addition rules is equivalent to a parallel application of rules (note that $n_r = r + \cdots (n) \cdots + r$).

3 Addition rule of evolution of a region

Now we are going to describe in a formal manner how objects evolve in a region.

3.1 Useful rules determination.

Let R_i the set of evolution rules associated to a determined labelled region i. The sub-set of rules which their consequent have all their destination labelled with adjacent region to i, is named useful rules set.

3.2 Active potentially rules determination.

Active potentially rules set of a region is formed removing from useful rules set those rules which their antecedent is not included in the multiset associated to the region.

3.3 Active rules determination. Maximal respect to priority.

Active rules set of a region is the maximal set respect partial order relationship described by priorities associated to the region over active potentially rules set of the region.

3.4 Set of all multiset over active rules.

Let A the set of all multiset over the active rules set in a region R. Let $A+$ the set formed by applying addition rules over every element from A. Let $B+$ the set formed removing from $A+$ those rules with their antecedent is not included in the multiset associated to R. Let $C+$ the maximal set in relation to antecedent inclusion of evolution rules over $B+$.

The set describing every possible parallel application of evolution rules in a region is formed by removing from A, those elements which their associated addition rule is not included in $C+$.

3.5 Satisfactory computation concept.

In a Transition P System, we will say that computation has finished satisfactorily if and only if:

i. There are no objects in the system which can evolve.

ii. In the system exists the output membrane i_0, and it is an elemental membrane.

4 Representation of Transition P System in silico

When we try to translate an algebraic structure into data structures, it is possible to find several possibilities due to:

i. The same data structure with different defined operation on itself can represent different algebraic structures.

ii. The same algebraic structure can be represented by different combinations of data structures with operations.

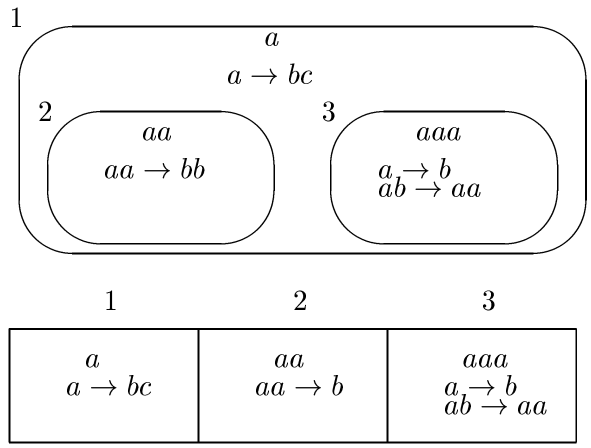


Figure 5: Representations with Independent References.

These facts make that main difference, between two data structures and their associated operations representing the same algebraic structure, will be efficiency in run time and allocated memory.

However, the best choice in order to represent an algebraic structure into a data structure will be the one that allows the highest efficiency maintaining the original algebraic structure properties.

Algebraic structure used for defining Transition P System has, from a computational point of view, one basic characteristic: hierarchical structure of regions and their contents are independents, but both of them are linked by labels. So, these facts make easy to design a structure of references (tree, graph) that automatically build a structure for referred items. That is to say, we can deal with items relationships (inter-structure) independently of themselves (intra-structure)

Firstly, we are going to present data structures based on these ideas. These kinds of structures are called *independents references.* Afterwards, we will present a data structure which no need refer region in order to define its structure, but it builds directly a region structure. It is called *region tree.*

4.1 Representations with Independent References

We need a data structure that permits to store regions and references to regions. Moreover, by efficiency aspects, access to regions must be direct or with time access complexity $O(1)$. Natural data structure satisfying these constrictions is *array* or *vector.* Main array structure disadvantage is the necessity of defining its dimension usually in compilation time. Moreover, it is a program constant without possibility of change in run-time. This data structure has a fixed size (static memory). Of course, it is possible to simulate dynamism, but without pass over the maximal permitted size. Finally, to say that array structure is efficient in memory because it represents region only one time, figure 5.

4.2 Labels Arboreal Structure Representation

How it has been said, a tree, algebraically speaking, can be represented by multiple data structures, computationally speaking. Among them:

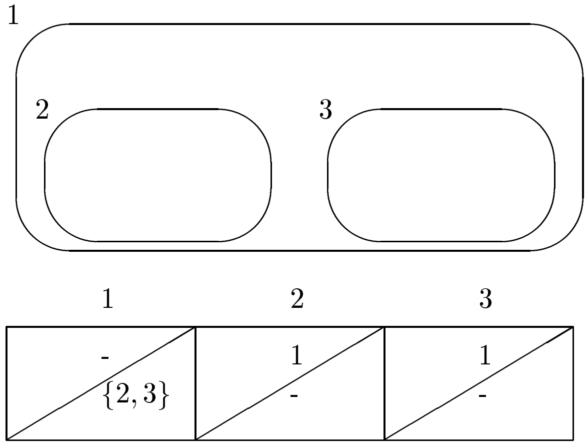


Figure 6: Vector representation with father's information.

i. An array or vector of pair of labels in the form (fathers label, sons label).

ii. A list of pair of labels in the form (fathers label, sons label).

iii. A square matrix whose index will be the labels.

However, those which permit an more efficient treatment due to they minimise unnecessary searches are the following:

i. An array or vector of pair in the form (fathers label, list of sons labels) and its index are labels. The main advantage of this data structure is that it permits a direct access from every region to its father region. That is why it represents a bi-directional tree. The structure has a certain redundancy degree, because the same label could appears as index, as father of some others ones or even as son of another one , figure 6.

ii. A general non-empty tree whose elements are labels. In this case labels appear only one time on the structure. This is the most likely structure in form to original Transition P Systems definition (figure 7).

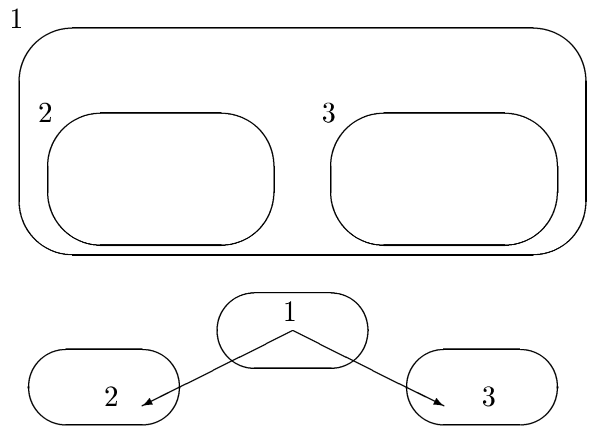


Figure 7: Tree representation with membrane index.

4.3 Representation as Regions Tree

How in a tree structure, labels appear only one time, so we can directly substitute labels by regions without loosing memory efficiency. That is to say the tree structure is used to store directly regions, not references to regions. In our opinion, this is the best way for representing P Systems, from a functional or declarative framework, figure 8.

5 Parallelism Treatment in Silico

How it has been said in theoretical explanation, Transition P Systems have two parallelism levels. Level one of parallelism or Global, and level two of parallelism or regional. This paper is not dealing with regional parallelism at a computational point of view; it is out of our purposes here. However, global parallelism is very important at this level of representation, since it will be necessary to maintain auxiliary data structures in order to store objects from each region. This allows us to simulate global parallelism in a sequential device.

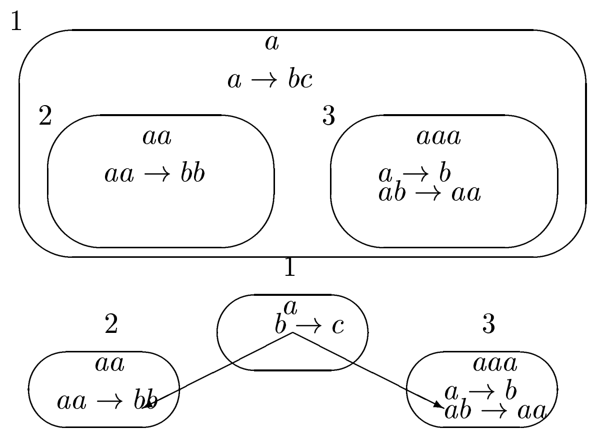


Figure 8: Tree representation with dynamic components.

Two processes can be executed in parallel if they are independents. So if we serialize the execution of independent parallel process, we can execute each one of them in a non-deterministic order. This fact gives a great freedom degree when we are simulating in sequential silicon media a parallelism biological based process like Transition P Systems.

To maintain a certain order in the global parallelism simulation process, we decided to perform a depth search throught the region tree.

Let us explain what happens in this process: When we firstly access to a region of the tree, it is possible that in the simulation process some objects from the region evolve transitting to adjacents regions. If target region has not been simulated yet, it means that objects sent to this region cannot be incorporated to its associated multiset R_i. So these objects must be store in an auxiliary multiset. Second time we access to a region, auxiliary object multiset is included in the associated multiset region.

In some cases, it could be possible that membrane containing a region must be disolve. If it would be disolved the first time we visit the region (it could be happened that objects from inner adjacent regions evolved transitting into this region) then it cannot be done because it could be some objects from inner adjacent regions which could evolve to the region and has already been disolved. So they (objects) have no target region. What this fact means is that some evolution rules are useful no more.

Therefore to avoid this case we will disolve membranes second time we access to regions.

6 Conclusions

P Systems are a very versatile computational model due to their capability for modifying their structure in run time [5, 6]. They are also very powerful computational devices because of their high parallelism degree.

To simulate global paralelism behaviour of transition P Systems in a sequential computational device it has been required to include auxiliary data structures from original model.

References

[1] Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61 (2000), and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).

[2] Gh. Păun, G. Rozenberg, A. Salomaa, Membrane computing with external output, *Fundamenta Informaticae*, 41, 3 (2000), 259–266, and *Turku Center for Computer Science-TUCS Report* No 218, 1998 (www.tucs.fi).

[3] Gh. Păun, Computing with membranes (P systems): Twenty six research topics, *Auckland University, CDMTCS Report* No 119, 2000 (www.cs.auckland.ac.nz/CDMTCS).

[4] J. Castellanos, G. Paun, Rodriguez-Paton A, *Computing with Membranes: P Systems with Worm-Objects*. IEEE Conf. SPIRE 2000, and Auckland University, CDMTCS Report No. 123, 2000 (www.cs.auckland.ac.nz/CDMTCS).

[5] S. N. Krishna, R. Rama, On the power of P systems with sequential and parallel rewriting, *International J. of Computer Math.*, in press.

[6] J. Dassow, Gh. Păun, On the power of membrane computing, *J. of Universal Computer Sci.*, 5, 2 (1999), 33–49 (www.iicm.edu/jucs).

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 35 - 50.

Visual Multiset Rewriting[1]

Paolo Bottoni,[2] Bernd Meyer[2,3] and Francesco Parisi Presicce[2]

[2]Dipartimento di Scienze dell' Informazione
Universita' La Sapienza di Roma
[3]School of Computer Science & Software Engineering
Monash University, Australia

[bottoni | parisi]@dsi.uniroma1.it
bernd.meyer@acm.org

Abstract

Diagrammatic notations, such as Venn diagrams, Petri-Nets and finite state automata, are in common use in mathematics and computer science. While the semantic domain of such systems is usually well formalized, the visual notation itself almost never is, so that they cannot be used as valid devices of formal reasoning. A complete formalization of such notations requires the construction of diagram systems with rigorously defined syntax and semantics. We discuss how diagram specification can be interpreted as multiset rewriting and, based on this, how it can be formalized in linear logic. The main contribution of the paper is the identification of a small fragment of linear logic that is sufficiently expressive for modelling diagram parsing and transformation, yet small enough to be suitable for automated deduction. We prove the equivalence of our modelling to attributed multiset grammar approaches and demonstrate its utility with two concrete applications: Animation of finite state automata and diagrammatic reasoning about sets with Venn diagrams.

1 Motivation

Diagrammatic notations in mathematics date as far back as Euler circles, Venn diagrams, Peirce's existential graphs and Frege's Beweisschrift. Computer science uses many diagrammatic notations of its own, such as Petri-Nets or finite state automata. While the underlying abstract semantics of such systems is usually well formalized, the visual notation itself almost never is. Hence, even though "execution" of diagrams is often used as a simulation technique for reasoning about the system behavior, it is not possible to treat the visual level of this reasoning process formally.

Consequentially, most researchers reject visual methods as valid devices of formal reasoning. A major line of contemporary research in diagrammatic reasoning therefore attempts to establish the status of diagrams as reasoning devices that are acceptable in proofs and other formal mathematical arguments. Obviously this

[1]This work is in part supported by the European Community TMR network GETGRATS.

requires the construction of diagram systems with rigorously defined syntax and semantics as well as the construction of visual proof calculi on the basis of which an analysis of the soundness and completeness of these systems can be given. The prototypical example of diagram systems for which this has successfully been performed are variants of Venn diagrams [Shi95, Ham96]. In a software engineering context, such a formalization could be the basis for a rigorous treatment of tasks like refinement and consistency checks using diagrammatic notations like UML.

The diversity of types and applications of diagrammatic notations has led to a variety of individual specialized formalization techniques, typically based on rewriting systems. In this paper we interpret diagram transformations as consumption and production of graphical tokens and demonstrate how a modelling as multiset rewriting opens the way for a unified treatment on the basis of linear logic.

The main contribution of the paper is the identification of a small fragment of linear logic sufficient for modelling diagram parsing and transformation, yet small enough to be suitable for automated deduction. We show how certain forms of diagrammatic reasoning can be formalized in this fragment with two concrete applications: Animation of finite state automata and diagrammatic reasoning about sets with Venn diagrams.

2 Representing Diagrams as Multisets

The first step to formalizing diagram transformation systems obviously is to find a proper representation for static diagrams. Diagrams can be considered as collections of graphical tokens such as *line*s, *circle*s, etc. A fundamental distinction between diagrammatic languages and string languages is that no single linear order is inherently defined on diagram elements. Hence, diagrammatic expressions must be considered as non-linear structures, such as graphs or multisets, rather than as sequences of tokens. Diagram transformation can then be interpreted as rewriting of such structures. In this paper we describe a multiset-based approach, i.e. we view the tokens as resources that can be produced, consumed, queried and changed. Linear logic presents itself naturally as a framework for studying diagram transformations based upon such a model.

The objects in a multiset that represents a diagram must contain a representation of their geometry. Obviously, a proper typing mechanism is also desirable. For example, on the most elementary level, the square of Figure 1 could be represented as the multiset $\{line(140, 5), line(140, 85), line(220, 85), line(220, 5)\}$. However, we might want to model objects on a higher level of abstraction, so that the entire square in the figure is represented as a single object $\{square(140, 5, 80)\}$. The construction of this new type from the primitives could be defined by a simple rule and parsing could be used to construct the higher-level presentation.

For the purpose of this article, we adopt a view in which meaningful types of graphical entities are given by specific graphical data types. We regard their construction as encapsulated in abstract constructors and thus the process by which they can be recognized is hidden. From a technical perspective, the recognition of these data types from a diagram can be done in various forms, for example by transformation of prototypes [BCM99], by parsing [MMW98], with algebraic specifications [BMST99] or using description logic [Haa98].

A strong requirement to be imposed on the graphical data type system is that type assignment must be unambiguous. For example, in a system where both a *rectangle* and a *square* data type exist, rectangles with equal sides must consistently be considered as squares. Inheritance can be used to model the relation between these types [Haa98].

An important requirement for the transformation mechanism is implied: A transformation which modifies the geometry of an element so that it is incompatible with its original data type must also change the type of this element accordingly. In the rest of this paper we assume that these conditions are met by the type systems and by the transformation rules adopted in the examples. For discussions of the limits of these assumptions see [BCM99, BMST99, MMW98].

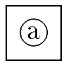


Figure 1: A simple diagram.

The second question to be addressed is how to represent the spatial relations between the diagrammatic objects. Several alternatives for this have been considered in the literature:

Explicit representation of relations: The relations existing between tokens are reified and inserted in the multiset as tokens in themselves. In this case, the diagram of Figure 1 could be represented as $\{letter_1(\text{"}a\text{"}, 35, 45, 10, 10), square_1(0,0,80), circle_1(40,40,20), inside(letter_1, circle_1), inside(circle_1, square_1)\}$.

Use of constraints: The multiset is combined with a set of constraints that describes the relevant spatial relations expressed as (usually arithmetic) constraints on attributes of the objects. The pair (M, C) representing the diagram of Figure 1 could be described by $\{letter(\text{"}a\text{"}, \vec{x}_1), circle(\vec{x}_2), square(\vec{x}_3)\}$ together with the constraints $\{inside(\vec{x}_1, \vec{x}_2) \wedge inside(\vec{x}_2, \vec{x}_3)\}$. Since attributes are used to represent the geometries, spatial relations can be queried by inspection of this geometry, e.g. by testing constraint entailment or consistency. These functions are realized by a constraint solver for the underlying domain.

Use of concrete geometries: For this representation, tokens with geometric attributes are used as in the case of the constraint representation. However, attributes are always given concrete values. In this case, the representation of Figure 1 would be $\{letter(\text{"}a\text{"}, 35, 45, 10, 10), square(0, 0, 80), circle(40, 40, 20)\}$.

The choice of representation has some important implications related to the frame problem. Consider the transformation rule depicted in Figure 2 which expresses the removal of a *circle* object. It corresponds to a multiset transformation rule which could informally be written as:

$$circle(\_,\_,\_), square(X,Y,Z), inside(circle, square) \rightarrow square(X,Y,Z)$$

Applying this rule to Figure 1 we are left with the problem of what to do with the *inside* relation between the letter and the circle. Since the circle no longer exists, it should be removed, but now an *inside* relation should hold between the letter and the square. This causes problems with the first two representations.

The fundamental difference in the way these representations handle the problem is this: In the first representation, the *inside* relation is represented explicitly by an uninterpreted, symbolic relation. Any transformation must therefore explicitly handle such relations and relevant change in the spatial relations must be propagated explicitly. Essentially, the specification needs to re-implement the relevant fragment of geometry, which makes the specification cumbersome and error-prone.

In the constraint-based representation, the *inside* relation is managed by mapping the geometry to the underlying arithmetic domain and by having a constraint solver handle the arithmetic constraint theory. The propagation of spatial constraints happens for free, because the corresponding arithmetic constraints are propagated automatically by this solver. However, in the

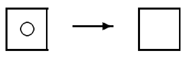


Figure 2: Simple Transformation.

context of *transformation* we are now facing a new problem, which stems from the fact that an appropriate constraint solver must work incrementally. The full set of constraints is only evaluated once for the initial diagram. After this, the constraint store should be adapted *incrementally* during the transformation, i.e. only those spatial constraints that change should be updated, added or deleted. The problem arises, because constraints in the store have to be kept in a solved form, so that it is difficult to perform a meaningful constraint deletion. Even if a single constraint can explicitly be removed, it is difficult to remove all implied constraints. As a simple example, consider the existence of three attributes a, b, c. When the constraints $a = b \wedge b = c$ are asserted, $a = c$ is automatically derived. Even if we remove $b = c$ from the constraint store, $a = c$ would still be in the store, but it would no longer be justified. Transformation of constraint diagram representations therefore requires a re-evaluation of the existing spatial relations, which is inefficient. These problems have been explored in [Mey97].

These kinds of problem make it difficult to provide a linear logic characterization of diagram transformation based on either of these representations. For this reason, we adopt the representation based on concrete geometry here, and we assume the existence of a specialized *geometry agent* which is able to answer queries regarding spatial relations in a given diagram based on its concrete geometry.

3 Diagram Transformation as Multiset Rewriting

We now move from discussing the representation of diagrams as multisets to investigating diagrammatic transformations through multiset transformations. In general, such diagram transformations detail how one diagram can *syntactically* be transformed into another diagram that directly and explicitly exhibits additional information that is either not present or only implicit and hidden in the prior diagram. This transformation can be understood as multiset transformation or rewriting. This is not a completely new idea, since whereas the specification of diagram transformation has often been based on graph grammars [BMST99], other approaches have also used *type-0* variants of attributed multiset grammars [MMW98, MM00].

Even though different formal theories for multi-dimensional grammars have been developed, there is no grammar calculus that would allow us to deduce soundness or completeness results for diagrammatic calculi on the basis of their grammar specifications. It is for these reasons that logic has been explored as an alternative tool for diagram rewriting. The first obvious choice to explore is classical first-order logic. Roughly speaking, two different embeddings are possible: Either the objects of the diagram and their relations are represented as predicates or, alternatively, they can be modelled as term structures. For diagram rewriting, the first type of embedding has been demonstrated in [HM91, Mar94, Mey00]. The second approach is closely related to modelling string language grammars in logic programming by Definite Clause Grammars and to their extension in the form of Definite Clause Set Grammars [Tan91]. Variations for the use of diagram parsing and/or rewriting have been demonstrated and discussed in [HMO91, Mey97, MMW98].

Both possible embeddings of diagram rewriting into first-order logic have drawbacks that make their universal utility questionable. In the first embedding (graphical entities as predicates) a typical rewrite step needs to add as well as to remove objects which amounts to deriving new predicates (conclusions) and deleting old conclusions. This is not possible in classical first order logic due to its monotonicity. In the second embedding there are no restrictions on how the diagram could be rewritten, but the modelling does not leverage from the structure of the underlying logic anymore, since, essentially, this is "abused" as a rule-based rewrite mechanism. In contrast, what is really desirable is that the logical derivation relation can directly deal with terms representing the graphical elements, so that a direct correspondence between logical derivations and those in the diagram system exists.

A detailed analysis of these problems has recently been given for the first time in [MM00], where the use of linear logic [Gir87, Gir91] is advocated instead. The major advantage of linear logic over classical logic in the context of diagram rewriting is that it is a logic of resources and actions, and therefore adequately models a non-monotonous rewrite process. Linear implication models the fact that the left-hand side resources (the antecedent) are used and consumed (!) in the process of producing the right-hand side resources (the consequent). This is exactly the process of multiset rewriting that we have to model.

The question arises which fragment of linear logic we should use. Obviously we need to use multiplicative connectives to express the simultaneous existence of elements in the multiset of diagram elements and linear implication is adequate to express the rewriting as such. Since *all* the elements in a multiset of objects representing a diagram have to exist simultaneously, a natural choice is to use multiplicative conjunction ($\otimes$) to model their union in a multiset. Therefore, we might decide to express, for instance, the transformation rule in Figure 2 as

$$circle(\vec{x}_1) \otimes square(\vec{x}_2) \otimes inside(\vec{x}_1, \vec{x}_2) \multimap square(\vec{x}_2)$$

This choice, advocated in [MM00], seems natural and conceptually correct. However, we are not interested in modelling diagram rewriting in linear logic for its theoretical properties alone, but we are also interested in linear logic as a well-founded theoretical framework for declarative computational implementations of diagram transformation. Therefore, we have to pay attention to whether the chosen fragment is adequate as the basis of implementations.

3.1 LO and Interaction Abstract Machines

The basic idea for an implementation of our framework is that it should be directly transformable into a linear logic program. Unfortunately, current linear logic programming languages [Mil95] do not offer multiplicative conjunction in the rule head. It would therefore be advantageous to find a different fragment of linear logic that directly corresponds to a linear logic programming language. The fragment we introduce in this paper is a slight extension of the one used in the linear logic programming language LO. A benefit of LO is that it has an interpretation as a system of interacting collaborating systems of agents, called Interaction Abstract Machine (IAM). This can later be used as the basis of integrating interaction specifications with our approach. We will now give a very brief introduction to the IAM and its interpretation in LO. For a full introduction the interested reader is referred to [ACP93, AP91].

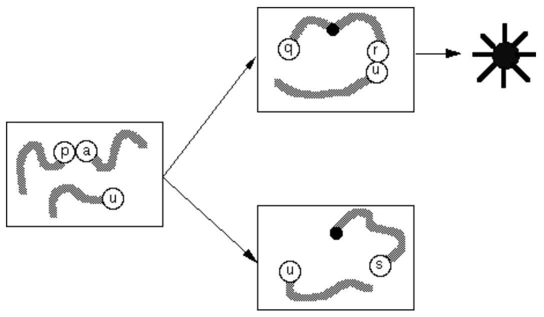


Figure 3: IAM Evolution

The IAM is a model of interacting agents in which an agent's state is fully described as a multiset of resources. The agents' behavior is described by a set of rules called "methods". A method takes the form $A_1 \parr \cdots \parr A_n \multimapinv B$, meaning that a multiset containing the elements $A_1, \cdots, A_n$ (the *head*) can be transformed by eliminating these elements and producing effects as specified by the form of B (the *body*). Each resource A_i and each resource in B is syntactically given as a first-order term $f(X_1, \ldots, X_n)$ like in standard logic programming. A method can fire if the appropriate head elements are found among the agent's resources.

If more than one rule can fire at a given time, e.g. if the agent's resource set is $\{a, p, q\}$ and its rules are $p \parr a \multimapinv r$ and $q \parr a \multimapinv s$, these rules compete, i.e. one of them will be chosen, committed and applied. If the body could only contain the connective $\parr$, the rules would only allow us to describe the behavior of a single agent, but IAMs also use a second kind of connective (in the body of a rule) which effectively spawns another agent. All the resources of the spawning agent not consumed by the rule application are copied into each spawned agent. This connective is the $\&$, read "with". $\&$ takes priority on $\parr$. A final IAM operation is to terminate an agent. This is denoted by a $\top$ in the body of a method. This behavior is illustrated in Figure 3 (redrawn from [ACP93]), depicting a possible development of an agent with state $\{p, a, u\}$ whose behavior is specified by the rules $p \parr a \multimapinv (q \parr r) \ \& \ s$ and $r \parr u \multimapinv \top$.

The complete behavior of a method in a multiagent IAM is summarized by the following rules: (1) A rule is applicable to an agent only if all the rule head atoms occur in the agent's state. (2) If a rule is applicable and is selected for application to an agent, the rule's head atoms are first removed from the agent's state; (3) The new configuration of the IAM is defined according to the form of the body: (a) If the body is the symbol $\top$, the agent is terminated; (b) If the body is the symbol $\perp$, no atom is added to the state of the agent; (c) If the body does not contain any $\&$, then the body elements are added to the agent state; (d) If the body consists of several conjuncts connected by $\&$, then for each occurrence of $\&$ a new agent containing a copy of the original agent's resources is spawned. For all the resulting agents (including the original one) the atoms in the corresponding conjunct are added to the agent's state.

The IAM, even when restricted to the single agent case, defines a form of multiset rewriting that is applicable to the rewriting of diagrams as outlined above.

The advantage gained from using the IAM as our basic model is that its interpretation can be given in a small fragment of linear logic which only consists of the connectives *par* ($\parr$), *with* ($\&$) and linear implication ($\multimapinv$), as implemented by the linear logic programming language LO [AP91]. IAMs and LO also use another kind of connective that emulates broadcasting among different agents. We do not need this connective for our tasks and will therefore exclude it.

3.2 Diagram Parsing as Linear Logic Programming

Parsing can be considered as the most basic task in diagram transformation. First, it seems of fundamental importance to be able to analyze the correctness of a diagram and to interpret its structure. Secondly, context-free parsing according to a multiset grammar corresponds to a particularly elementary form of diagram transformation in which each transformation step replaces a multiset of diagram objects with some non-terminal object. We will therefore first look at diagram parsing in linear logic, before proceeding to arbitrary diagram transformation. Diagram parsing has been studied by a number of different researchers before. The interested reader is referred to [MMW98, BMST99] for comprehensive surveys and to [MM00] for diagram parsing in linear logic. Here we review a particular type of attributed multiset grammars, termed constraint multiset grammars (CMGs), which have been used by a number of researchers for reasonably complex tasks such as the interpretation of state diagrams and mathematical equations. In [Mar94] a precise formal treatment is given, and we review only the basic notions here. CMG productions rewrite multisets of attributed tokens and have the form

$$U ::= U_1, \ldots, U_n \ where \ (C) \ \{E\} \tag{1}$$

indicating that the non-terminal symbol U can be recognized from the symbols U_1, ..., U_n whenever the attributes of U_1, ..., U_n satisfy the constraints C. The attributes of U are computed using the assignment expression E. The constraints enable information about spatial layout and relationships to be naturally encoded in the grammar. The terms *terminal* and *non-terminal* are used analogously to the case in string languages. The only difference lies in the fact that terminal types in CMGs refer to graphic primitives, such as *line* and *circle*, instead of textual tokens and each of these *symbol types* has a set of one or more attributes, typically used to describe its geometric properties. A *symbol* is an instance of a symbol type. In each grammar, there is a distinguished non-terminal symbol type called the *start* type.

CMGs also include context-sensitive productions. Context symbols, i.e. symbols that are not consumed when a production is applied, are existentially quantified in a production. As an example, the following context-sensitive production from a grammar for state transition diagrams recognizes transitions:

T:transition ::= A:arc exist S1:state,S2:state where (
OnCircle(A.start,S1.mid,S1.radius) and OnCircle(A.end,S2.mid,S2.radius))
{T.start = S1.label and T.tran = A.label and T.end = S2.label }

A diagrammatic sentence to be parsed by a CMG is just an attributed multiset of graphical tokens. Therefore we can view a sentential form as the resources of an IAM agent. Intuitively, it is clear that the application of a CMG production corresponds closely to the firing of IAM methods and that a successful parse consists of rewriting the original set of terminal symbols into a multiset that only contains the single non-terminal symbol which is the start symbol.

We can map a CMG production to an LO rule (IAM method, respectively) and hence to a linear logic implication in the following way: For a CMG production

$$U ::= U_1, \ldots, U_n \ exists \ U_{n+1}, \ldots, U_m \ where \ (C) \ \{E\} \tag{2}$$

we use the equivalent LO rule:

$$\tau(u_1) \parr \ldots \parr \tau(u_m) \multimapinv \{C\} \& \{E\} \& \tau(u) \parr \tau(u_{n+1}) \parr \ldots \parr \tau(u_m) \tag{3}$$

In LO, each CMG terminal and non-terminal object u_i will be represented by a first order term $\tau(u_i)$ which has the token type of u_i as the main functor and contains the attributes in some fixed order. We extend this mapping function canonically so that we use $\tau(u_1, \ldots, u_n)$ to denote $\tau(u_1) \parr \ldots \parr \tau(u_n)$. In the same way, $\tau(p)$, for a CMG production p, will denote its mapping to an LO rule and $\tau(G) = \{\tau(p) \mid p \in P\}$ denotes the complete mapping of a CMG G to an LO program.

The linear logic reading of such a rule $\tau(p)$ is its exponential universal closure:

$$! \widetilde{\forall} \ \tau(u_1) \parr \ldots \parr \tau(u_m) \multimapinv \{C\} \& \{E\} \& \tau(u) \parr \tau(u_{n+1}) \parr \ldots \parr \tau(u_m) \tag{4}$$

To evaluate the constraints in the grammar and to compute the attribute assignments we assume that the geometric (arithmetic) theory is available as a first-order theory Γ_g in linear logic. Obviously, geometry cannot be completely axiomatized in

LO, because its fragment of linear logic is too small. However, we can encapsulate a more powerful "geometry machine" (and arithmetic evaluator) in a separate agent and give evaluation requests to this agent. This is what we do by using "with" (&) to spawn agents for these computations in the above LO translation.

From an operational point of view, this requires us to adopt a proactive interpretation of LO in which we can spawn an agent and wait for it to return a result before proceeding with the rest of the computation. A different implementation of a proactive LO interpretation, by sending requests from a coordinator to registered participants, is provided by the Coordination Language Facility [AFP96].

Each rule $\tau(p)$ emulates exactly one production p. To emulate parsing fully, we also need a rule which declares that a parse is successful if and only if the initial diagram is reduced to the start symbol and no other symbols are left. For a CMG G with start symbol s, we could do this in linear logic by adding $\tau(s)$ as an axiom to $\tau(G)$. Unfortunately, from an implementation point of view, we cannot formulate true linear axioms in LO. It is more consistent with the LO model to extend the language with the linear goal $\mathbf{1}$, which terminates an agent if and only if this agent does not have any resources left (i.e. $\mathbf{1}$ succeeds if and only if the linear proof context is empty). We will call this extension of the LO language LO_1. Instead of the axiom $\tau(s)$ we can then add the method $\tau(s)\circ\!-\mathbf{1}$ to the LO_1 program. The complete set of LO_1 rules that implement a grammar G is: $\Pi = \tau(G) \cup \{(\tau(s)\circ\!-\mathbf{1})\}$

Operationally, a successful parse of a diagram D now corresponds to an IAM evolution with method set Π starting from a single agent with the resource set $\tau(D)$ in which all agents eventually terminate. Logically, it corresponds to a proof of $\Gamma_g, \Pi \vdash \tau(D)$. This linear logic embedding of CMGs is sound and complete.

Theorem 1 $D \in \mathcal{L}(G) \Leftrightarrow \Gamma_g, \Pi \vdash \tau(D)$

The proof is given in the appendix.

LO_1 is only a minor extension of LO and a proper subset of the linear logic programming language Lygon [HPW96]. Thus we still have an executable logic as the basis of our model. In fact, it is Lygon's sequent calculus [HP94] that we will use in the remainder of this article. In contrast to LO which applies rules by committed choice, Lygon actually performs a search for a proper proof tree. Therefore, if there is a proof for $\Gamma_g, \Pi \vdash \tau(D)$, i.e. if $D \in \mathcal{L}(G)$, Lygon will find this proof. This is in contrast to LO, which, even disregarding the extension with $\mathbf{1}$, can only be guaranteed to find the proof if G is confluent.

4 Applications

In the previous sections we have presented a general approach to diagram transformation based on linear logic. In this section we present two simple applications of this approach: One in which the diagram transformation corresponds to the execution of a computation and one in which diagram transformations are used to reason in some underlying domain represented by the diagram.

4.1 Executable diagrams

By executable diagrams we refer to such diagram notations that are used to specify the configurations of some system. Transformation of such diagrams can be used to simulate and animate the transformation of these configurations. Typical examples of such systems are Petri nets, finite state machine and a number of domain specific visual languages. For these systems, a multiset representation is intuitively apt and the transformation rules for the multiset closely correspond to the diagram transformation rules. As an example, consider a transition in a finite state diagram, such as the one in Figure 4.

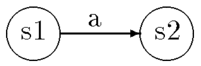


Figure 4: FSA Transition

Let us adopt a set of data types corresponding to the definition of *state*s, with a geometry, a name and a couple of attributes denoting whether the state is initial or final; *transition*s, defined by their geometry and an input symbol; and *input* labels, which are positioned under the current state and are read one symbol at a time.

A straightforward translation of the partial diagram of Figure 4 results in a rule which corresponds exactly to the semantics of the depicted transition, assuming that in a diagrammatic animation of the transformation the input string is always placed under the current state:

$$\begin{aligned}&\mathit{state}((109,24), s1, \mathit{nonfinal}, \mathit{noninitial}) \parr \mathit{transition}((129,24),(189,24), \text{“}a\text{”}) \parr\\ &\quad \mathit{state}((209,24), s2, \mathit{nonfinal}, \mathit{noninitial}) \parr \mathit{input}((109,40), [\text{“}a\text{”}|\mathit{Rest}])\\ &\quad \multimapinv\ \mathit{state}((109,24), s1, \mathit{nonfinal}, \mathit{noninitial}) \parr \mathit{transition}((129,24),(189,24), \text{“}a\text{”}) \parr\\ &\qquad \mathit{state}((209,24), s2, \mathit{nonfinal}, \mathit{noninitial}) \parr \mathit{input}((209,40), \mathit{Rest})\end{aligned}$$

The whole diagram is translated into such a set of rules, one for each transition, and its execution can be started by placing the input string under the initial state.

We can, however, have a more general view of this process and define the behavior of such animations independently of a concrete diagram:

$$\begin{aligned}&\mathit{state}(\mathit{Geom}1, \mathit{Name}1, F1, I1) \parr \mathit{state}(\mathit{Geom}2, \mathit{Name}2, F2, I2) \parr\\ &\quad \mathit{transition}(\mathit{Geom}3, \mathit{Lab}) \parr \mathit{input}(\mathit{Geom}4, [\mathit{Lab}|\mathit{Rest}])\\ &\quad \multimapinv\ \mathit{state}(\mathit{Geom}1, \mathit{Name}1, F1, I1) \parr \mathit{state}(\mathit{Geom}2, \mathit{Name}2, F2, I2) \parr\\ &\qquad \mathit{transition}(\mathit{Geom}3, \mathit{Lab}) \parr \mathit{input}(\mathit{Geom}5, \mathit{Rest})\ \&\\ &\qquad \mathit{startsat}(\mathit{Geom}3, \mathit{Geom}1)\ \&\ \mathit{endsat}(\mathit{Geom}3, \mathit{Geom}2)\ \&\\ &\qquad \mathit{below}(\mathit{Geom}4, \mathit{Geom}1)\ \&\ \mathit{below}(\mathit{Geom}5, \mathit{Geom}2)\end{aligned}$$

where *startsat, endsat* and *below* are suitable predicates that check the corresponding spatial relations, possibly instantiating the *geom* attribute appropriately.

In the LO_1 setting, a rule for expressing acceptance of the input would be:

$$\mathit{state}(\mathit{Geom}1, \mathit{Name}1, \mathit{final}, \_) \parr \mathit{input}(\mathit{Geom}2, []) \multimapinv \top \& \mathit{below}(\mathit{Geom}2, \mathit{Geom}1).$$

Note that the termination of an agent indicates the success of a branch in the corresponding proof.

It is easy to see how the two alternative approaches can both provide an operational semantics for executable diagrams. In both cases, the actual execution of the transformations occurs uniformly according to the LO_1 proof system.

4.2 Diagrammatic reasoning

Often we are using diagram transformations not so much to define the configuration of a computational system, but instead to reason about some abstract domain. A typical case of the use of diagrams to perform such reasoning are Venn Diagrams. A variant of these, developed by Shin [Shi95], provides a formal syntax, semantics and a sound and complete system of visual inference rules.

In these diagrams, sets are represented by regions, shaded regions imply that the corresponding set is empty and a chain of X implies that at least one of the regions marked by an X in the chain must be non-empty.

As an example, Figure 5 says that A is non-empty (expressed by the chain of X), nothing is both in A and in B (expressed by shading), and at least one element is in B. By inference we can obtain that the elements in B and in A must belong to the symmetric difference $A - B \cup B - A$. This diagram is equivalent to one in which the X in the shaded region is removed. Such an equivalence is expressed by the "Erasure of Links" inference rule. This can be stated as "an X in a shaded region may be removed from an X-chain provided the chain is kept connected." We reformulate this textual rule as a set of linear logic rules, defined on the following graphical data types: (1) *chain*, associated with an attribute *setOfX* which stores the locations of the X elements in the chain; (2) x, with an attribute *pt* giving its position and an attribute *num*, giving the number of lines attached to it; (3) *line*, with an attribute *ends* giving the positions of its two ends; (4) *region*, with an attribute *geom*, allowing the reconstruction of the geometry of the region, and an attribute *shading*, indicating whether the region is shaded or not. Additionally, some synchronization resources are used to indicate that the transformation is performing some not yet completed process. Link erasure is defined by the following actions:

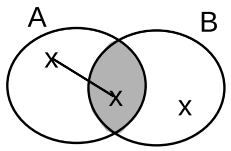


Figure 5: A Venn diagram

(1) A point inside a shaded region is eliminated and the set in the chain is updated accordingly. A synchronization resource is used to ensure that all the elements previously connected to it will be considered:

$$\begin{aligned}&chain(SofX) \parr x(Pt, Num) \parr region(Geom, shaded)\\ &\quad \multimapinv chain(G, Pt, Num) \parr region(Geom, shaded) \;\&\\ &\qquad inside(Pt, Geom) \;\&\; G == SofX \setminus \{Pt\}\end{aligned}$$

(2) Points previously connected to the removed element are marked and the connecting lines are removed:

$$\begin{aligned}&chain(G, Pt, Num) \parr line(Ends) \parr x(Pt1, Num1)\\ &\quad \multimapinv chain(G, Pt, Num) \parr x(Pt1, cand, Num11) \;\&\\ &\qquad Ends == \{Pt, Pt1\} \;\&\; Num11 = Num1 - 1\end{aligned}$$

(3) If the removed point was inside the chain, its two neighbors are connected by a new line. Synchronization resources are removed and a consistent state is restored:

$$\begin{aligned}&chain(G, Pt, 2) \parr x(Pt1, cand, Num1) \parr x(Pt2, cand, Num2)\\ &\quad \multimapinv chain(G) \parr x(Pt1, Num11) \parr x(Pt2, Num21) \parr line(Ends) \;\&\\ &\qquad Ends == \{Pt1, Pt2\} \;\&\; Num11 = Num1 + 1 \;\&\; Num21 = Num2 + 1\end{aligned}$$

(4) If the removed point was at an end of the chain, its neighbor is now at an end.

$$chain(G, Pt, 1) \parr x(Pt1, cand, 1) \multimapinv chain(G) \parr x(Pt1, 1)$$

(5) If the removed point was an isolated element, the diagram was inconsistent, and the chain is removed altogether:

$chain(G, Pt, 0) \multimapinv \bot$

The erasure process goes through intermediate steps in which the diagram is not a Venn diagram (for instance, dangling edges appear in the chain). Such inconsistent diagrams correspond to states in which synchronization resources appear in the multiset. The process is, however, guaranteed to terminate with a consistent diagram. Such situations often occur in diagrammatic transformations, where a complex step is broken up to produce several intermediate diagrams, of which only the start and final diagram belong to the language of interest. The problem of deciding whether a diagram produced during the transformation process belongs to the language or is just an intermediate diagram can in many cases be solved without resorting to parsing. In fact, knowing that the starting diagram was in the language and knowing the possible transformations, we can usually define some simple test for the validity of a transformed diagram. For example, among all the multisets produced during the link erasure process, all and only those which do not contain any synchronization resource represent a Venn diagram.

In general, a language L can be specified by a triple $(L_0, \Rightarrow^*, L_f)$, where L_0 is an initial language, $\Rightarrow^*$ is the reflexive and transitive closure of the *yield* relation, and L_f is a final language acting as a filter for sentences produced starting from L_0 according to $\Rightarrow^*$, i.e. $L = \{s \mid \exists s' \in L_0 : s' \Rightarrow^* s\} \cap L_f$.

This view was proposed for string languages in [Man98] and independently adopted for the diagrammatic case in [BCM99, BPPS00]. This suggests a line of attack for typical problems in diagram transformations related to the possible production of inconsistent diagrams. In our approach, the filter language can be characterized by a set of LO_1 rules. A valid state in a diagram transformation process is one for which there exists an LO_1 proof of the filter property.

As an example, consider the *dangling edge* problem which is typical of graph transformation systems. The double-pushout approach to algebraic graph transformation [CMR+97] faces this problem by not allowing deletion of a node if its elimination would leave dangling edges after rule application. From our perspective, this could be modelled by giving a simple set of LO_1 filter rules:

$edge(G1) \parr node(G2) \parr node(G3) \multimapinv node(G2) \parr node(G3) \& touches(G1, G2) \& touches(G1, G3)$
$node(\_) \multimapinv \bot$
$node(\_) \multimapinv \mathbf{1}$

5 Conclusions

We have shown how diagram transformation can be formalized in linear logic and we have discussed interpretations in multiset rewriting. Many important kinds of diagrammatic reasoning, which can be understood as syntactic diagram transformation, can be formalized in this way.

The main technical contribution of this paper over previous work is the identification of a small fragment of linear logic that is expressive enough to model diagrammatic transformations, yet small enough to directly correspond to a calculus of linear logic programming. Our formalism therefore is a directly executable specification language. We have also proven equivalence of our model with attributed multiset grammar approaches and correctness of the corresponding mapping.

The next extension to be investigated is negative application conditions. These are required in many transformation systems to check the non-existence of certain contexts or to ensure exhaustive rule application. It is not yet clear whether LO_1 is an adequate and sufficiently strong fragment of linear logic to model such systems.

From an implementation point of view, it appears worthwhile to explore the integration of constraints into linear logic programming languages.

Ultimately, we are interested in specification languages for diagram notations in which the rules themselves are visual. The idea is that this can be formalized by an additional mapping between linear logic rules and visual rules. Such an approach necessarily raises the question if and when visual rules are adequate to describe a transformation system. We hope that the ability to formalize the transformation as well as the embedding conditions and the underlying geometric theory within the unifying framework of linear logic will allow us to develop formal criteria that help to answer this important question.

Appendix A: Linear Sequent Calculus

This appendix shows the relevant rules of the sequent calculus presented in [HP94].

$$\frac{}{\phi \vdash \phi}\ (ax) \qquad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma', \phi \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ (cut)$$

$$\frac{\Gamma, \phi, \psi, \Gamma' \vdash \Delta}{\Gamma, \psi, \phi, \Gamma' \vdash \Delta}\ (X-L) \qquad \frac{\Gamma \vdash \Delta, \phi, \psi, \Delta'}{\Gamma \vdash \Delta, \psi, \phi, \Delta'}\ (X-R)$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \psi \& \phi \vdash \Delta} \qquad \frac{\Gamma, \psi \vdash \Delta}{\Gamma, \psi \& \phi \vdash \Delta}\ (\&-L) \qquad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \& \psi, \Delta}\ (\&-R)$$

$$\frac{\Gamma, \phi \vdash \Delta \quad \Gamma', \psi \vdash \Delta'}{\Gamma, \Gamma', \psi \parr \phi \vdash \Delta, \Delta'}\ (\parr - L) \qquad \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \parr \psi, \Delta}\ (\parr - R)$$

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma', \psi \vdash \Delta'}{\Gamma, \Gamma', \phi \multimap \psi \vdash \Delta, \Delta'}\ (\multimap -L) \qquad \frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}\ (\forall - L)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ (W! - L) \qquad \frac{\Gamma, !\phi, !\phi \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ (C! - L)$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ (! - L) \qquad \frac{}{\vdash \mathbf{1}}\ (\mathbf{1} - R)$$

Appendix B: Proof of Theorem 1

Due to space restrictions we can only give a limited amount of detail here. We first show the "only if" direction. An accepting derivation in G has the following structure: $D \rightarrow_{p_{i_1}} D_1 \rightarrow_{p_{i_2}} \rightarrow \ldots \rightarrow_{p_{i_n}} \{s\}$ In which $\rightarrow_{p_{i_j}}$ indicates the application of production p_{i_j} in step j. We show that each derivation step j corresponds to a valid sequent in linear logic. We can consider each derivation step in isolation. Let p_{i_j} have the form (2). Then derivation step j has the form: $\{V, u_1, \ldots, u_m\} \rightarrow_{p_{i_j}} \{V, u, u_{n+1}, \ldots, u_m\}$ where V is the application context, $u_{n+1}, \ldots, u_m$ is the rule context and there is a ground substitution θ for C and E such that $\Gamma_g \vdash (C \wedge E)\theta$ where Γ_g is the geometric/arithmetic theory. Let $\Upsilon = \tau(V)$, $v = \tau(u)$, $v_i = \tau(u_i)$.

Now, the linear equivalent of p_{i_j} is the exponential universal closure of $\tau(p_{i_j})$ which has the form (4). Therefore the following sequent can be constructed:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \begin{array}{c}
            \dfrac{
              \begin{array}{c}
                \dfrac{
                  \dfrac{}{\Gamma_g \vdash \top, \Upsilon}\,(\top - R)
                }{} \\
                \vdots
              \end{array}
            }{\Gamma_g \vdash (C\&E)\theta, \Upsilon}
            \qquad
            \Gamma_g, \Pi \vdash v\theta \parr v_{n+1} \parr \ldots \parr v_m, \Upsilon
          \end{array}
        }{\Gamma_g, \Gamma_g, \Pi \vdash (C\&E)\theta \& v\theta \parr v_{n+1} \parr \ldots \parr v_m, \Upsilon}\,(\&-R)
        }{
        \begin{array}{c} \vdots \end{array}
      }
      \over
      \dfrac{}{\Gamma_g, \Pi \vdash (C\&E)\theta \& v\theta \parr v_{n+1} \parr \ldots \parr v_m, \Upsilon}\,(C! - L)
      \qquad *_1
    }{\Gamma_g, \Pi, \tau(p_{i_j})\theta \vdash v_1, \ldots, v_m, \Upsilon}\,(\multimap - L)
  }{
    \dfrac{\Gamma_g, \Pi, \widetilde{\forall}\tau(p_{i_j}) \vdash v_1, \ldots, v_m, \Upsilon}{\Gamma_g, \Pi \vdash v_1, \ldots, v_m, \Upsilon}\,(C! - L); (! - L)
  }\,(\forall - L)
}{
  \begin{array}{c} \vdots \\ \hline \Gamma_g, \Pi \vdash v_1 \parr \ldots \parr v_m, \Upsilon \end{array}
}\,(\parr - R)
$$

$$
\dfrac{
  \begin{array}{c}
    \dfrac{
      \dfrac{}{v_1 \vdash v_1}\,(ax) \quad \ldots \quad \dfrac{}{v_m \vdash v_m}\,(ax)
    }{} \\
    \vdots
  \end{array}
}{v_1 \parr \ldots \parr v_m \vdash v_1 \parr \ldots \parr v_m}\,(\parr - L)
\qquad
\begin{array}{c} \\ *_1 \end{array}
$$

Therefore, to prove that $\Gamma_g, \Pi \vdash D_i$ it suffices to show that $\Gamma_g, \Pi \vdash D_{i+1}$. So all that remains to show is that $\Gamma_G, \Pi \vdash \tau(s)$. This is trivial, since we have included the appropriate rule $\tau(s) \multimap \mathbf{1}$ explicitly in Π:

$$
\dfrac{
  \dfrac{
    \begin{array}{c}
      \dfrac{
        \dfrac{}{\vdash \mathbf{1}}\,(1 - R)
      }{} \\
      \vdots
    \end{array}
  }{\Gamma_g, \Pi \vdash \mathbf{1}}\,(W! - L)
  \qquad
  \dfrac{}{\tau(s) \vdash \tau(s)}\,(ax)
}{\Gamma_g, \Pi \vdash \tau(s)}\,(\multimap - L)
$$

In the opposite direction ("if") the proof proceeds by induction on the number of derivation steps in the linear proof. We first have to note that every linear representation of a sentential form has a special syntactic form:[2] In $\Gamma \vdash \Delta$, the linear sentential form representation Δ on the right hand side must be of a form corresponding to some $\tau(D)$. This is the case if and only if $\Delta = \Upsilon$ or $\Delta = C \& v_0 \parr \Upsilon$, where v_0 is a token corresponding to a terminal or non-terminal symbol, $\Upsilon = v_1 \parr \ldots \parr v_n$ is a multiplicative disjunction of tokens and $C = C_0 \& \ldots \& C_m$ is an additive conjunction of arithmetic/geometric constraints, i.e. C does not contain any tokens. Υ can also take the form $\Upsilon = v_1, \ldots, v_n$ which we consider as an alternative linear representation for the sentential form $v_1 \parr \ldots \parr v_n$.

We will show that every proof that ultimately leads to a conclusion $\Gamma \vdash \Delta$ in which Δ is in this form contains only sequents of the forms

$$\dfrac{}{\Gamma \vdash \Delta} \tag{5}$$

$$\dfrac{\Gamma_1 \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2}{\Gamma \vdash \Delta} \tag{6}$$

[2]Note that subsequently we will use the terms *sentential form* and *linear representation of a sentential form* interchangeably where the intended meaning is evident from the context.

in which the left hand side can be decomposed as $\Gamma = \Gamma_g, \Pi, \Sigma$ into arithmetic axioms Γ_g, the grammar rules Π and a multiset of tokens Σ and Δ is a sentential form that can be derived from Σ according to Π under the theory Γ_g. Note that we consider $\Delta = C_1 \& \ldots \& C_k$ with $\Gamma_g \vdash \Delta$ as a sentential form for an empty diagram and that the empty diagram is implicitly always contained in the language.

Throughout the proof, the left hand side Γ of any sequent can only be augmented except for by application of $(\multimap -L)$. But $(\multimap -L)$ introduces a form $\phi \multimap \psi$ into Γ which must be the representation of a grammar rule, since no other implications may ultimately exists on the left hand side. Therefore ψ must be of the form $\Sigma = \psi_1 \parr \ldots \parr \psi_m$.

This means that only axioms of the geometric theory and rules for the grammar productions or elements of the form of Σ may be introduced into Γ in any sequent for the proof to finally arrive at the form $\Gamma_g, \Pi \vdash \tau(D)$ where $\Pi = \{\tau(G), \tau(s) \multimap \mathbf{1}\}$. It follows that the left hand side of any sequent in the proof can be decomposed as $\Gamma = \Gamma_g, \Pi, \Sigma$ into arithmetic axioms Γ_g, the grammar Π and a multiset of tokens Σ.

W.l.o.g. we assume that the arithmetic/geometric theory Γ_g contains all arithmetic truths as facts, i.e. contains no implications. We also note that we can replace the rule $\tau(s) \multimap \mathbf{1}$ in Π by the axiom $\tau(s)$ thus eliminating the single use of $\mathbf{1}$.

According to the syntactic structure of our rules, we can only have sequents of the following types in the proof after the elimination of cuts: $(ax), (X-L), (X-R), (\&-R), (\parr - L), (\parr - R), (\multimap -L), (!-L), (W!-L), (C!-L), (\forall - L)$.

Therefore any production of form (5) is of type (ax) so that Δ is a sentential form. If the proof contains only a single sequent, it must be of the form (5). Therefore Δ is a sentential form.

If the proof contains $n+1$ sequents, the last sequent can have any of the forms $(X-L), (X-R), (\&-R), (\parr-L), (\parr-R), (\multimap -L), (!-L), (W!-L), (C!-L), (\forall-L)$

The induction is trivial for $(X-L), (X-R)$, because only the order of elements in the grammar and axiom set (sentential form, respectively) is changed. The induction is also trivial for $(!-L), (W!-L), (C!-L)$ since only axioms and grammar rules are exponential. For $(\parr - R)$ it is trivial, because we consider ϕ, ψ, Δ and $\phi \parr \psi, \Delta$ as equivalent representations of the same sentential form. Thus we need only show that the induction holds for $(\&-R), (\parr - L), (\multimap -L)$.

In the case of

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \& \psi, \Delta}\ (\&-R)$$

we can observe that either ϕ or ψ must be an arithmetic/geometric truth, because otherwise $\Gamma \vdash \phi \& \psi, \Delta$ could never reach the syntactical structure required for a sentential form. Let this be ϕ. Then ψ must either be a token or an arithmetic/geometric truth and Δ must be a sentential form. So $\tau^{-1}(\psi, \Delta)$ is a sentential form that can be derived from Σ with the grammar $\tau^{-1}(\Pi)$ and $\Gamma_g \vdash \phi$, i.e. ϕ can be derived from the arithmetic theory. Therefore $\tau^{-1}(\phi \& \psi, \Delta)$ is a sentential form that can be derived from Σ with the grammar $\tau^{-1}(\Pi)$ under the axiom set Γ_g. This proves the induction for $(\&-R)$.

The form

$$\frac{\Gamma, \phi \vdash \Delta \quad \Gamma', \phi' \vdash \Delta'}{\Gamma, \Gamma', \phi \parr \phi' \vdash \Delta, \Delta'}\ (\parr - L)$$

is explained by the concatenation of two grammars: As above, we can decompose the left hand sides into arithmetic axioms Γ_g, the grammar Π and a multiset of tokens Σ (Γ'_g, Π', Σ', respectively). Thus the grammar $\tau^{-1}(\Pi)$ allows to derive $\tau^{-1}(\Delta)$

from $\tau^{-1}(\Sigma)$ under Γ_g and the grammar $\tau^{-1}(\Pi')$ allows to derive $\tau^{-1}(\Delta')$ from $\tau^{-1}(\Sigma')$ under Γ'_g. We can concatenate these grammars into a grammar G and the arithmetic theories into a theory T such that G allows to derive $\tau^{-1}(\Delta') \cup \tau^{-1}(\Delta)$ from $\tau^{-1}(\Sigma') \cup \tau^{-1}(\Sigma)$ under T. This proves the induction for $(\invamp - L)$.

For the case of

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma', \phi' \vdash \Delta'}{\Gamma, \Gamma', \phi \multimap \phi' \vdash \Delta, \Delta'} \; (\multimap -L)$$

we can decompose Γ, Γ' as above.

The grammar $\tau^{-1}(\Pi)$ allows to derive $\tau^{-1}(\phi, \Delta)$ from $\tau^{-1}(\Sigma)$ under Γ_g and the grammar $\tau^{-1}(\Pi')$ allows to derive $\tau^{-1}(\Delta')$ from $\tau^{-1}(\Sigma', \phi')$ under Γ'_g. We can concatenate these grammars into a grammar G and the arithmetic theories into a theory T such that G allows to derive $\tau^{-1}(\Delta') \cup \tau^{-1}(\Delta)$ from $\tau^{-1}(\Sigma), \tau^{-1}(\Sigma')$ under T, if we add the production $\tau^{-1}(\phi) ::= \tau^{-1}(\phi')$ to G. Exactly the linear representation of this production is added to the axiom set by $(\multimap -L)$.

This concludes the proof. □

References

[ACP93] J.-M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction abstract machines. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 257–280. MIT Press, Cambridge, MA, 1993.

[AFP96] J.-M. Andreoli, S. Freeman, and R. Pareschi. The coordination language facility: Coordination of distributed objects. *Theory and Practice of Object Systems*, 2:77–94, 1996.

[AP91] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[BCM99] P. Bottoni, M.F. Costabile, and P. Mussio. Specification and dialogue control of visual interaction through visual rewriting systems. *ACM Transactions on Programming Languages and Systems*, 21:1077–1136, 1999.

[BMST99] R. Bardohl, M. Minas, A. Schürr, and G. Taentzer. Application of graph transformation to visual languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 105–180. World Scientific, 1999.

[BPPS00] P. Bottoni, F. Parisi Presicce, and M. Simeoni. From formulae to rewriting systems. In H. Ehrig, G. Engels, H.-J. Kreowsky, and G. Rozenberg, editors, *Theory and Application of Graph Transformations*, pages 267–280. Springer, Berlin, 2000.

[CMR$^+$97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Lowe. Algebraic approaches to graph transformation - Part I: basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, pages 163–245. World Scientific, 1997.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir91] J.-Y. Girard. Linear logic: A survey. Technical report, Int. Summer School on Logic and Algebra of Specification, 1991.

[Haa98] V. Haarslev. A fully formalized theory for describing visual notations. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 261–292. Springer, New York, 1998.

[Ham96] E. Hammer. Representing relations diagrammatically. In G. Allwein and J. Barwise, editors, *Logical Reasoning with Diagrams*. Oxford University Press, New York, 1996.

[HM91] R. Helm and K. Marriott. A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2:311–331, 1991.

[HMO91] R. Helm, K. Marriott, and M. Odersky. Building visual language parsers. In *ACM Conf. Human Factors in Computing*, pages 118–125, 1991.

[HP94] J. Harland and D. Pym. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.

[HPW96] J. Harland, D. Pym, and M. Winikoff. Programming in Lygon: An overview. In *Algebraic Methodology and Software Technology*, LNCS 1101, pages 391–405. Springer, July 1996.

[Man98] V. Manca. String rewriting and metabolism: A logical perspective. In G. Paun, editor, *Computing with Bio-Molecules*, pages 36–60. Springer-Verlag, Singapore, 1998.

[Mar94] K. Marriott. Constraint multiset grammars. In *IEEE Symposium on Visual Languages*, pages 118–125. IEEE Computer Society Press, 1994.

[Mey97] B. Meyer. Formalization of visual mathematical notations. In M. Anderson, editor, *AAAI Symposium on Diagrammatic Reasoning (DR-II)*, pages 58–68, Boston/MA, November 1997. AAAI Press, AAAI Technical Report FS-97-02.

[Mey00] B. Meyer. A constraint-based framework for diagrammatic reasoning. *Applied Artificial Intelligence*, 14(4):327–344, 2000.

[Mil95] D. Miller. A survey of linear logic programming. *Computational Logic*, 2(2):63–67, December 1995.

[MM00] K. Marriott and B. Meyer. Non-standard logics for diagram interpretation. In *Diagrams 2000: International Conference on Theory and Application of Diagrams*, Edinburgh, Scotland, September 2000. Springer. To appear.

[MMW98] K. Marriott, B. Meyer, and K.B. Wittenburg. A survey of visual language specification and recognition. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 5–85. Springer, 1998.

[Shi95] S.-J. Shin. *The Logical Status of Diagrams*. Cambridge University Press, Cambridge, 1995.

[Tan91] T. Tanaka. Definite clause set grammars: A formalism for problem solving. *Journal of Logic Programming*, 10:1–17, 1991.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 51 - 67.

Rewriting and Multisets in ρ-calculus and ELAN

Horatiu Cirstea & Claude Kirchner
LORIA and INRIA and UHP
615, rue du Jardin Botanique
54600 Villers-lès-Nancy Cedex, France
{Horatiu.Cirstea,Claude.Kirchner}@loria.fr

Abstract

The ρ-calculus is a new calculus that integrates in a uniform and simple setting first-order rewriting, λ-calculus and non-deterministic computations. The main design concept of the ρ-calculus is to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule *application* and *multisets of results*. This paper describes the calculus from its syntax to its basic properties in the untyped case. The ρ-calculus embeds first-order conditional rewriting and λ-calculus and it can be used in order to give an operational semantics to the rewrite based language ELAN. We show how the set-like data structures are easily represented in ELAN and how this can be used in order to specify the Needham-Schroeder public-key protocol.

Keywords: Rewriting, Strategy, Multisets, Matching.

1 Introduction

It is a common claim that rewriting is ubiquitous in computer science and mathematical logic. And indeed the rewriting concept appears from the very theoretical settings to the very practical implementations. Some extreme examples are the mail system under Unix that uses rules in order to rewrite mail addresses in canonical forms (see the `/etc/sendmail.cf` file in the configuration of the mail system) and the transition rules describing the behaviors of a tree automata. Rewriting is used in semantics in order to describe the meaning of programming languages as well as in program transformations like, for example, re-engineering of Cobol programs [vdBvDK$^+$96]. It is used in order to compute [Der85], implicitly or explicitly like in Mathematica [Wol99] or OBJ [GKK$^+$87], but also to perform deduction when describing by inference rules a logic [GLT89], a theorem prover [JK86] or a constraint solver [JK91]. It is of course central in systems making the notion of rule an explicit and first class object, like expert systems, programming languages based on equational logic, algebraic specifications, functional programming and transition systems.

In this very general picture, we introduce a calculus whose main design concept is to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule *application* and *multisets of results*. We concentrate on *term* rewriting, we introduce a very general notion of rewrite rule and we make the rule application and result explicit concepts. These are the basic ingredients of the *rewriting-* or *ρ-calculus* whose originality comes from the fact that terms, rules, rule application and therefore rule application strategies are all treated at the object level.

In ρ-calculus we can explicitly represent the application of a rewrite rule (say $a \to b$) to a term (like the constant a) as the object $[a \to b](a)$ which evaluates to the singleton $\{b\}$. This means that the rule application symbol $@$ (where @ is our notation for the placeholder) is part of the calculus syntax.

But the application of a rewrite rule may fail like in $[a \to b](c)$ that evaluates to the empty set $\emptyset$ or it can be reduced to a multiset with more than one element like exemplified later in this section and explained in Section 2.3. Of course, variables may be used in rewrite rules like in $[f(x) \to x](f(a))$. In this last case the evaluation mechanism of the calculus will reduce the application to $\{a\}$. In fact, when evaluating this expression, the variable x is bound to a via a mechanism classically called matching, and we recover the classical way term rewriting is acting.

Where this game becomes even more interesting is that $@ \to @$, the rewrite arrow operator, is also part of the calculus syntax. This is a powerful abstractor whose relationship with λ-abstraction [Chu40] could provide a useful intuition: A λ-expression $\lambda x.t$ could be represented in the ρ-calculus as the rewrite rule $x \to t$. Indeed the β-redex $(\lambda x.t\ u)$ is nothing else than $[x \to t](u)$ (i.e. the application of the rewrite rule $x \to t$ on the term u) which reduces to $\{\{x/u\}t\}$ (i.e. the application of the substitution $\{x/u\}$ to the term t).

So, basic ρ-calculus objects are built from a signature, a set of variables, the abstraction operator $@ \to @$, the application operator $@$, and we consider multisets of such objects. That gives to the ρ-calculus the ability to handle non-determinism in the sense of multisets of results. This is achieved via the explicit handling of reduction result multisets, including the empty set that records the fundamental information of rule application failure. For example, if the symbol $+$ is assumed to be commutative then applying the rule $x + y \to x$ to the term $a + b$ results in $\{a, b\}$. Since there are two different ways to apply (match) this rewrite rule modulo commutativity the result is a set that contains two different elements corresponding to two possibilities.

To summarize, in ρ-calculus abstraction is handled via the arrow binary operator, matching is used as the parameter passing mechanism, substitution takes care of variable bindings and results multisets are handled explicitly.

The operational semantics of ELAN, a language based on labeled conditional rewrite rules and strategies controlling the rule application, can be described using the ρ-calculus. We use the ELAN language in order to describe and analyze the Needham-Schroeder public-key protocol [NS78].The implementation in ELAN is very concise and the rewrite rules describing the protocol are directly obtained from a classical presentation like the one given in Section 3.2.1.

2 Description of the ρ_T-calculus

We assume given in this section a theory T defined equationally or by any other means and we present the components of the ρ_T-calculus and we comment our main choices.

2.1 Syntax of the ρ_T-calculus

The *syntax* makes precise the formation of the objects manipulated by the calculus as well as the formation of substitutions that are used by the evaluation mechanism. In the case of ρ_T-calculus, the core of the object formation relies on a first-order signature together with rewrite rules formation, rule application and multisets of results.

Definition 2.1 We consider $\mathcal{X}$ a set of variables and $\mathcal{F} = \bigcup_m \mathcal{F}_m$ a set of ranked function symbols, where for all m, $\mathcal{F}_m$ is the subset of function symbols of arity m. We assume that each symbol has a unique arity i.e. that the $\mathcal{F}_m$ are disjoint. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the set of first-order terms built on $\mathcal{F}$ using the variables in $\mathcal{X}$.

The set of basic ρ-terms can thus be inductively defined by the following grammar:

$$\rho\text{-terms} \quad t \quad ::= \quad x \mid f(t, \dots, t) \mid \{t, \dots, t\} \mid [t](t) \mid t \to t$$

where $x \in \mathcal{X}$ and $f \in \mathcal{F}$.

We adopt a very general discipline for the rewrite rule formation, and we do not enforce any of the standard restrictions often used in the rewriting community like non-variable left-hand-sides or occurrence of the right-hand-side variables in the left-hand-side. We also allow rewrite rules containing rewrite rules as well as rewrite rule application. We consider that the symbols $\{\}$ and $\emptyset$ both represent the empty set. For the terms of the form $\{t_1, \dots, t_n\}$ we assume as usual that the comma is associative and commutative.

The main intuition behind this syntax is that a rewrite rule is an abstractor, the left-hand-side of which determines the bound variables and some contextual structure. Having new variables in the right-hand-side is just the ability to have free variables in the calculus. We will come back to this later but to support the intuition let us mention that the λ-terms and standard first-order rewrite rules [DJ90, BN98] are clearly objects of this calculus. For example, the λ-term $\lambda x.(y\ x)$ corresponds to the ρ-term $x \to [y](x)$ and a rewrite rule in first-order rewriting corresponds to the same rewrite rule in the rewriting-calculus.

We have chosen multisets as the data structure for handling the potential non-determinism. A multiset of terms could be seen as the set of distinct results obtained by applying a rewrite rule to a term. Other choices could be made depending on the intended use of the calculus. For example, if we do not want to provide the identical results of an application a set could be used. When the order of the computation of the results is important, lists could be employed. The confluence properties are similar in a the set and multiset approaches. It is clear that for the list approach only a confluence modulo permutation of lists can be obtained.

Example 2.1 If we consider $\mathcal{F}_0 = \{a, b, c\}$, $\mathcal{F}_1 = \{f, g\}$, $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1$ and x, y variables in $\mathcal{X}$, some ρ-terms from $\varrho(\mathcal{F}, \mathcal{X})$ are:

- $[a \to b](a)$; this denotes the application of the rewrite rule $a \to b$ to the term a. We will see that the evaluation of this application is $\{b\}$.
- $[f(x,y) \to g(x,y)](f(a,b))$; a classical rewrite rule application leading to a $\{g(a,b)\}$result.
- $[y \to [x \to x+y](b)]([x \to x](a))$; a ρ-term that corresponds to the λ-term $(\lambda y.((\lambda x.x+y)\ b))\ ((\lambda x.x)\ a)$.
- $[[(x \to x+1) \to (1 \to x)](a \to a+1)](1)$; a more complicated ρ-term without corresponding standard rewrite rule or λ-term.

These examples show the very expressive syntax that is allowed for ρ-terms.

2.2 Matching and substitution application

The *matching algorithm* is used to bind variables to their actual values. In the case of ρ_T-calculus, this is in general higher-order matching. But in practical cases it will be higher-order-pattern matching, or equational matching, or simply syntactic matching and their combination. The matching theory is specified as a parameter (the theory T) of the calculus and when it is clear from the context this parameter is omitted.

Definition 2.2 For a given theory T over ρ-terms, a *T-match-equation* is a formula of the form $t \ll^?_T t'$, where t and t' are ρ-terms. A substitution σ is a solution of the T-match-equation $t \ll^?_T t'$ if $T \models \sigma(t) = t'$. A *T-matching system* is a conjunction of T-match-equations. A substitution is a solution of a T-matching system P if it is a solution of all the T-match-equations in P. We denote by $\mathbf{F}$ a T-matching system without solution. A T-matching system is called *trivial* when all substitutions are solution of it.
We define the function *Solution* on a T-matching system $\mathcal{S}$ as returning the set of all T-matches of $\mathcal{S}$ when $\mathcal{S}$ is not trivial and $\{\mathbb{ID}\}$, where $\mathbb{ID}$ is the identity substitution, when $\mathcal{S}$ is trivial.

Notice that when the matching algorithm fails (i.e. returns $\mathbf{F}$) the function *Solution* returns the empty set.

Example 2.2 If $\ll^?_\emptyset$ denotes a syntactic matching and $\ll^?_C$ a commutative matching then we have:

1. $a \ll^?_\emptyset b$ has no solutions, and thus $\mathcal{S}olution(a \ll^?_\emptyset b) = \emptyset$;
2. $f(x,x) \ll^?_\emptyset f(a,b)$ has no solution and thus $\mathcal{S}olution(f(x,x) \ll^?_\emptyset f(a,b)) = \emptyset$;

3. $a \ll^?_\emptyset a$ is solved by all substitutions, and thus $\mathcal{S}olution(a \ll^?_\emptyset a) = \{\mathbb{ID}\}$;
4. $f(x, g(x, y)) \ll^?_\emptyset f(a, g(a, b))$ has as solution the substitution $\sigma \equiv \{x/a, y/b\}$, and $\mathcal{S}olution(f(x, g(x, y)) \ll^?_\emptyset f(a, g(a, b))) = \{\sigma\}$;
5. $x + y \ll^?_C a + b$ has the two solutions $\{x/a, y/b\}$ and $\{x/b, y/a\}$ and thus $\mathcal{S}olution(x + y \ll^?_C a + b) = \{\{x/a, y/b\}, \{x/b, y/a\}\}$.

The description of the *substitution application* on terms is often given at the meta-level, except for explicit substitution frameworks.

As for any calculus involving binders like the λ-calculus, α-conversion should be used in order to obtain a correct substitution calculus and the first-order substitution (called here grafting) is not directly suitable for ρ-calculus. In order to obtain a substitution that takes care of variable bindings we consider the usual notions of α-conversion and higher-order substitution as defined for example in [DHK00].

The burden of variable handling could be avoided by using an explicit substitution mechanism in the spirit of [CHL96]. We sketched such an approach in [CK99a] and this will be detailed in a forthcoming paper.

2.3 Evaluation rules of the ρ_T-calculus

The *evaluation rules* describe the way the calculus operates. It is the glue between the previous components and the simplicity and clarity of these rules are fundamental for the calculus usability.

The evaluation rules of the ρ_T-calculus describe the application of a ρ-term on another one and specify the behavior of the different operators of the calculus when some arguments are multisets. They are defined in Figure 1.

In the rule *Fire*, $\{\sigma_1, \ldots, \sigma_i, \ldots\}$ represents the set of substitutions obtained by T-matching l on p (i.e. $\mathcal{S}olution(l \ll^?_T p)$) and $\sigma_i r$ represents the result of the application of the substitution σ_i on the term r. When the matching yields a failure represented by an empty set of substitutions, the result of the application of the rule *Fire* is the empty set.

We should point out that, like in λ-calculus an application can always be evaluated, but unlike in λ-calculus, the set of results could be empty. More generally, when matching modulo a theory T, the set of resulting matches may be empty, a singleton (like in the empty theory), a finite set (like for associativity-commutativity) or infinite (like for associativity). We have thus chosen to represent the result of a rewrite rule application to a term as a multiset. An empty set means that the rewrite rule $l \to r$ fails to apply on t in the sense of a matching failure between l and t.

In order to push rewrite rule application deeper into terms, we introduce the two *Congruence* evaluation rules. They deal with the application of a term of the form $f(u_1, \ldots, u_n)$ (where $f \in \mathcal{F}_n$) to another term of a similar form. When we have the same head symbol for the two terms of the application $[u](v)$ the arguments of the term u are applied on those of the term v argument-wise. If the head symbols are not the same, an empty set is obtained.

$$
\begin{array}{lll}
\mathit{Fire} & [l \to r](t) & \Longrightarrow \\
 & \{\sigma_1 r, \ldots, \sigma_n r, \ldots\} & \\
 & \multicolumn{2}{r}{\text{where } \sigma_i \in \mathcal{S}olution(l \ll^{?}_{T} t)} \\
\mathit{Congruence} & [f(u_1, \ldots, u_n)](f(v_1, \ldots, v_n)) & \Longrightarrow \\
 & \{f([u_1](v_1), \ldots, [u_n](v_n))\} & \\
\mathit{Congruence\_fail} & [f(u_1, \ldots, u_n)](g(v_1, \ldots, v_m)) & \Longrightarrow \\
 & \emptyset & \\
\mathit{Distrib} & [\{u_1, \ldots, u_n\}](v) & \Longrightarrow \\
 & \{[u_1](v), \ldots, [u_n](v)\} & \\
\mathit{Batch} & [v](\{u_1, \ldots, u_n\}) & \Longrightarrow \\
 & \{[v](u_1), \ldots, [v](u_n)\} & \\
\mathit{Switch}_L & \{u_1, \ldots, u_n\} \to v & \Longrightarrow \\
 & \{u_1 \to v, \ldots, u_n \to v\} & \\
\mathit{Switch}_R & u \to \{v_1, \ldots, v_n\} & \Longrightarrow \\
 & \{u \to v_n, \ldots, u \to v_n\} & \\
\mathit{OpOnSet} & f(v_1, \ldots, \{u_1, \ldots, u_m\}, \ldots, v_n) & \Longrightarrow \\
 & \{f(v_1, \ldots, u_1, \ldots, v_n), \ldots, f(v_1, \ldots, u_m, \ldots, v_n)\} & \\
\mathit{Flat} & \{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\} & \Longrightarrow \\
 & \{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\} &
\end{array}
$$

Figure 1: The evaluation rules of the ρ_T-calculus

The reductions corresponding to the cases where some sub-terms are multisets are defined by the last evaluation rules in Figure 1. These rules describe the propagation of the multisets on the constructors of the ρ-terms: the rules *Distrib* and *Batch* for the application, Switch_L and Switch_R for the abstraction and *OpOnSet* for functions. The evaluation rule that corresponds to the multiset propagation for set symbols and that eliminates the redundant set symbols is the evaluation rule *Flat*.

This design decision to use multisets to represent reduction results has another important consequence concerning the handling of sets with respect to matching. Indeed, sets are just used to store results and we do not wish to make them part of the theory. We are thus assuming that the matching operation used in the *Fire* evaluation rule is *not* performed modulo set axioms. This requires in some cases to use a strategy that pushes set braces outside the terms whenever possible.

To summarize, we can say that every time a ρ-term is reduced using the rules *Fire*, *Congruence* and *Congruence_fail* of the ρ_T-calculus, a multiset is generated. These evaluation rules are the ones that describe the application of a rewrite rule at the top level or deeper in a term. The multiset obtained when applying one of the above evaluation rules can trigger the application of the other evaluation rules of the calculus. These evaluation rules deal with the (propagation of) multisets and compute a "set-normal form" for the ρ-terms by pushing out the set braces and flattening the sets.

2.4 Evaluation strategies for the ρ_T-calculus

The *strategy* guides the application of the evaluation rules. The strategy $\mathcal{S}$ guiding the application of the evaluation rules of the ρ_T-calculus could be crucial for obtaining good properties for the calculus. In a first stage, the main property analyzed is the confluence of the calculus and if the rule *Fire* is applied under no conditions at any position of a ρ-term confluence does not hold.

The use of multisets for representing the reductions results is the main source of non-confluence. Unlike in the standard definition of a rewrite step where the rule application yields always a result, in ρ-calculus a rule application always yields a unique result that can be a multiset with several elements, representing the non-deterministic choice of the corresponding results from rewriting, or with no elements ($\emptyset$), representing the failure. Therefore, the relation generated by the evaluation rules of the ρ-calculus is finer and consequently non-confluent.

The confluence can be recovered if the evaluation rules of ρ-calculus are guided by an appropriate strategy. This strategy should first handle properly the problems related to the propagation of failure over the operators of the calculus. It should also take care of the correct handling of multisets with more than one element in non-linear contexts and details on this strategy are given in [CK99b].

2.5 Using the ρ_T-calculus

The aim of this section is to make concrete the concepts we have just introduced by giving a few examples of ρ-terms and ρ-reductions. Many other examples could be found on the ELAN web page [Pro00].

Let us start with the functional part of the calculus and give the ρ-terms representing some λ-terms. For example, the λ-abstraction $\lambda x.(y\ x)$, where y is a variable, is represented as the ρ-rule $x \to [y](x)$. The application of the above term to a constant a, $(\lambda x.(y\ x)\ a)$ is represented in the $\rho_\emptyset$-calculus by the application $[x \to [y](x)](a)$. This application reduces in the λ-calculus to the term $(y\ a)$ while in the $\rho_\emptyset$-calculus the result of the reduction is the singleton $\{[y](a)\}$. When a functional representation $f(x)$ is chosen, the λ-term $\lambda x.f(x)$ is represented by the ρ-term $x \to f(x)$ and a similar result is obtained. One should notice that for ρ-terms of this form (i.e. that have a variable as a left-hand side) the syntactic matching performed in the $\rho_\emptyset$-calculus is trivial, it never fails and gives only one result.

There is no difficulty to represent more elaborated λ-terms in the $\rho_\emptyset$-calculus. Let us consider the term $\lambda x.f(x)\ (\lambda y.y\ a)$ with the β-derivation: $\lambda x.f(x)\ (\lambda y.y\ a) \longrightarrow_\beta \lambda x.f(x)\ a \longrightarrow_\beta f(a)$. The same derivation can be recovered in the $\rho_\emptyset$-calculus for the corresponding ρ-term: $[x \to f(x)]([y \to y](a)) \longrightarrow_{Fire} [x \to f(x)](\{a\}) \longrightarrow_{Batch} \{[x \to f(x)](a)\} \longrightarrow_{Fire} \{\{f(a)\}\} \longrightarrow_{Flat} \{f(a)\}$. Of course, several reduction strategies can be used in the λ-calculus and reproduced accordingly in the $\rho_\emptyset$-calculus.

Now, if we introduce contextual information in the left-hand sides of the rewrite rules we obtain classical rewrite rules like $f(a) \to f(b)$ or $f(x) \to g(x)$. When we apply such a rewrite rule the matching can fail and consequently the application of the rewrite rule can fail. As we have already insisted in the previous sections, the failure of a rewrite rule is not a meta-property in the $\rho_\emptyset$-calculus but is represented by an empty set (of results). For example, in standard term rewriting we say that the application of the rule $f(a) \to f(b)$ to the term $f(c)$ fails while in the $\rho_\emptyset$-calculus the term $[f(a) \to f(b)](f(c))$ evaluates to $\emptyset$.

When the matching is done modulo an equational theory we obtain interesting behaviors. Take, for example, the list operator $\circ$ that appends two lists with elements of sort *Elem*. Any object of sort *Elem* represents a list consisting of this only object.

If we define the operator $\circ$ as right-associative, the rewrite rule taking the first part of a list can be written in the associative ρ_A-calculus $l \circ l' \to l$ and when applied to the list $a \circ b \circ c \circ d$ gives as result the ρ-term $\{a, a \circ b, a \circ b \circ c\}$. If the operator $\circ$ had not been defined as associative we would have obtained as result of the same rule application one of the singletons $\{a\}$ or $\{a \circ b\}$ or $\{a \circ (b \circ c)\}$ or $\{(a \circ b) \circ c\}$, depending of the way the term $a \circ b \circ c \circ d$ is parenthesized.

Let consider now a commutative operator $\oplus$ and the rewrite rule $x \oplus y \to x$ that selects one of the elements of the tuple $x \oplus y$. In the commutative ρ_C-calculus the application $[x \oplus y \to x](a \oplus b)$ evaluates to the set $\{a, b\}$ that represents the set of non-deterministic choices between the two results. The rewrite rule $x \oplus y \to x$ applies as well on the term $a \oplus a$ and the result is the multiset $\{a, a\}$ representing the non-deterministic choice between the two elements that in this case represents two possible reductions with the same result. In a set approach the result of this latter reduction is $\{a\}$.

We can also use an associative-commutative theory like, for example, when an operator describes multiset formation. Let us go back to the $\circ$ operator but this time let us define it as associative-commutative and use the rewrite rule $x \circ x \circ L \to L$ that eliminates doubletons from lists of sort *Elem*. Since the matching is done modulo associativity-commutativity this rule eliminates the doubletons no matter what is their position in the multiset. For instance, in the ρ_{AC}-calculus the application $[x \circ x \circ L \to L](a \circ b \circ c \circ a \circ d)$ evaluates to $\{b \circ c \circ d\}$: the search for the two equal elements is done thanks to associativity and commutativity.

Another facility is due to the use of multisets for handling non-determinism. This allows us to easily express the non-deterministic application of a multiset of rewrite rules on a term. Let us consider, for example, the operator $\otimes$ as a syntactic operator. If we want the same behavior as before for the selection of each element of the couple $x \otimes y$, two rewrite rules should be non-deterministically applied like in the reduction: $[\{x \otimes y \to x, x \otimes y \to y\}](a \otimes b) \longrightarrow_{Distrib} \{[x \otimes y \to x](a \otimes b), [x \otimes y \to y](a \otimes b)\}$ $\longrightarrow_{Fire} \{\{a\}, \{b\}\} \longrightarrow_{Flat} \{a, b\}$.

As we have seen, the ρ-calculus can be used for representing some simpler calculi like λ-calculus and rewriting. This can be proved formally by restricting the syntax and the evaluation rules of the ρ-calculus in order to represent the terms of the two calculi. Thus, for any reduction in the λ-calculus or conditional rewriting a corresponding natural reduction in the ρ-calculus can be found. We can extend the encoding of conditional rewriting in the ρ-calculus to more complicated rules like the conditional rewrite rules with local assignments from the ELAN language.

3 Specifications in the ELAN language

3.1 ELAN's rewrite rules

ELAN is an environment for specifying and prototyping deduction systems in a language based on labeled conditional rewrite rules and strategies to control rule application. The ELAN system offers a compiler and an interpreter of the language. The ELAN language allows us to describe in a natural and elegant way various deduction systems [BKK+96]. It has been experimented on several non-trivial applications ranging from decision procedures, constraint solvers, logic programming and automated theorem proving but also specification and exhaustive verification of authentication protocols [Pro00]. ELAN's rewrite rules are conditional rewrite rules with local assignments. The local assignments are let-like constructions that allow applications of strategies on some terms. The general syntax of an ELAN rule is:

$$[\ell] \quad l \Rightarrow r \qquad [\ \textbf{if} \ \ \mathit{cond} \ \ | \ \ \textbf{where} \ \ y := (S)u\ ]^* \quad \mathit{end}$$

We should notice that the square brackets ([ ]) in ELAN are used to indicate the label of the rule and should be distinguished from the square brackets of the ρ-calculus that represent the application of a rewrite rule (ρ-term).

The application of the labeled rewrite rules is controlled by user-defined strategies while the unlabeled rules are applied according to a default normalization strategy. The normalization strategy consists in applying unlabeled rules at any position of a term until the normal form is reached, this strategy being applied after each reduction produced by a labeled rewrite rule.

The application of a rewrite rule in ELAN can yield several results due to the equational (associative-commutative) matching and to the `where` clauses that can return as well several results.

Example 3.1 An example of an ELAN rule describing a possible naive way to search the minimal element of a list by sorting the list and taking the first element is the following:

```
[min-rule]  min(l)   =>   m
                 if l !=  nil
                 where sl := (sort) l
                 where m  := () head(sl)  end
```

The strategy `sort` can be any sorting strategy. The operator `head` is supposed to be described by a confluent and terminating set of unlabeled rewrite rules.

The evaluation strategy used for evaluating the conditions is a leftmost innermost standard rewriting strategy.

The non-determinism is handled mainly by two basic strategy operators: `dont care choose` (denoted `dc`$(s_1, \dots, s_n)$) that returns the results of at most one non-deterministicly chosen unfailing strategy from its arguments and `dont know choose`(denoted `dk`$(s_1, \dots, s_n)$) that returns all the possible results. A variant of the `dont care choose` strategy operator is the `first choose` operator (denoted `first`$(s_1, \ldots, s_n)$) that returns the results of the first unfailing strategy from its arguments.

Several strategy operators implemented in ELAN allow us a simple and concise description of user defined strategies. For example, the concatenation operator denoted `;` builds the sequential composition of two strategies s_1 and s_2. The strategy $s_1; s_2$ fails if s_1 fails, otherwise it returns all results (maybe none) of s_2 applied to the results of s_1. Using the operator `repeat*` we can describe the repeated application of a given strategy. Thus, `repeat*`(s) iterates the strategy s until it fails and then returns the last obtained result.

Any rule in ELAN is considered as a basic strategy and several other strategy operators are available for describing the computations. Here is a simple example illustrating the way the `first` and `dk` strategies work.

Example 3.2 If the strategy `dk(x => x+1,x => x+2)` is applied on the term a, ELAN provides two results: $a + 1$ and $a + 2$. When the strategy `first(x => x+1,x => x+2)` is applied on the same term only the $a + 1$ result is obtained. The strategy `first(b => b+1,a => a+2)` applied to the term a yields the result $a + 2$.

Using non-deterministic strategies we can explore exhaustively the search space of a given problem and find paths described by some specific properties.

A partial semantics could be given to an ELAN program using the rewriting logic [Mes92], but more conveniently ELAN's rules can be expressed using the ρ-calculus and thus an ELAN program is just a set of ρ-terms.

3.2 Representing multisets in ELAN

Using non-deterministic strategies we can explore exhaustively the set of states of a given problem and find paths described by some specific properties. For example, for proving the correctness of the Needham-Schroeder authentication protocol [NS78] we look for possible attacks among all the behaviors during a session.

In the this section we briefly present some of the rules of the protocol and we give the strategy looking for all the possible attacks, a more detailed description of the implementation is given in [Cir99].

3.2.1 The Needham-Schroeder public-key protocol

The Needham-Schroeder public-key protocol [NS78] aims to establish a mutual authentication between an initiator and a responder that communicate via an insecure network. Each agent A possesses a *public key* denoted $K(A)$ that can be obtained by any other agent from a key server and a *(private) secret key* that is the inverse of $K(A)$. A message m encrypted with the public key of the agent A is denoted by $\{m\}_{K(A)}$ and can be decrypted only by the owner of the corresponding secret key, i.e. by A.

The protocol uses *nonces* that are fresh random numbers to be used in a single run of the protocol. We denote the nonce generated by the agent A by N_A.

The simplified description of the protocol presented in [Low95] is:

1. $A \to B$: $\{N_A, A\}_{K(B)}$
2. $B \to A$: $\{N_A, N_B\}_{K(A)}$
3. $A \to B$: $\{N_B\}_{K(B)}$

The initiator A seeks to establish a session with the agent B. For this A sends a message to B containing a newly generated nonce N_A and its identity, message encrypted with its key $K(B)$. When such a message is received by the agent B, he can decrypt it and extract the nonce N_A and the identity of the sender. The agent B generates a new nonce N_B and he sends it to A together with N_A in a message encrypted with the public key of A. When A receives this response he can decrypt it and assumes that he has established a session with B. The agent A sends the nonce N_B back to B and when receiving this last message B assumes that he has established a session with A since only A could have decrypted the message containing N_B.

The main property expected for an authentication protocol like the Needham-Schroeder public-key protocol is to prevent an intruder from impersonating one of the two agents.

The intruder is an user of the communication network and so, he can initiate standard sessions with the other agents and he can respond to messages sent by the other agents. The intruder can intercept any message from the network and can decrypt the messages encrypted with its key. The nonces obtained from the decrypted messages can be used by the intruder for generating new (fake) messages. The intercepted messages that can not be decrypted by the intruder can be replayed as they are.

3.2.2 Encoding the Needham-Schroeder public-key protocol in ELAN

We present now a description of the protocol in ELAN. The ELAN rewrite rules correspond to transitions of agents from one state to another after sending and/or receiving messages.

Data structures The initiators and the responders are agents described by their identity, their state and a nonce they have created. An agent can be defined in ELAN using a mixfix operator:

```
@ + @ + @  : ( AgentId SWC Nonce ) Agent;
```

The symbol `@` is a placeholder for terms of types `AgentId`, `SWC` and `Nonce` respectively representing the identity, the state and the current nonce of a given agent.

There are three possible values of `SWC` states. An agent is in the state `SLEEP` if he has not sent nor received a request for a new session. In the state `WAIT` the agent has already sent or received a request and when reaching the state `COMMIT` the agent has established a session.

A nonce created by an agent A in order to communicate with an agent B is represented by `N(A,B)`. Memorizing the nonce allows the agent to know at each moment who is the agent with whom he is establishing a session and the two identities from the nonce are used when verifying the invariants of the protocol. A dummy nonce is represented by `N(di,di)`.

The nonces generated in the ELAN implementation are not random numbers but store some information indicating the agents using the nonce. If the uniqueness of nonces is important like, for example, in an implementation describing sequential runs of the protocol, an additional (random number) information can be easily added to the structure of nonces.

The agents exchange messages defined by:

```
@-->@:@[@,@,@] : (AgentId AgentId Key Nonce Nonce Address) message;
```

A message of the form `A-->B:K[N1,N2,Add]` is a message sent from `A` to `B` and contains the two nonces `N1` and `N2` together with the explicit address of the sender, `Add`. The address contains in fact the identity of the sender but we give it a different type in order to have a clear distinction between the identity of the sender in the encrypted part of the message and in the header of the message. The header of the message contains the source and destination address of the message but since they are not encrypted they can be faked by the intruder. The body of the message is encrypted with the key `K` and can be decrypted only by the owner of the private key.

The communication network is described by a possibly empty multiset of messages:

```
@       : ( message ) network;
@ & @   : ( network  network ) network (AC);
nill    : network;
```

with `nill` representing the network with no messages.

The intruder does not only participate to normal communications but can as well intercept and create (fake) messages. Therefore a new data structure is used for intruders:

```
@ # @ # @ : ( AgentId setNonce network ) intruder;
```

where the first field represents the identity of the intruder, the second one is the set of nonces he knows and the third one the set of messages he has intercepted. In our specification we only use one intruder and thus, the first field can be replaced by a constant identifying the intruder.

As for the messages, a set of nonces (`setNonce`) is defined using the associative-commutative operator `|` and a set of agents is defined using the associative-commutative operator `||`.

The ELAN rewrite rules are used to describe the modifications of the global state that consists of the states of all the agents involved in the communication and the state of the network. The global state is defined by:

```
@ <> @ <> @ <> @  :  ( setAgent setAgent intruder network ) state;
```

where the first two fields represent the set of initiators and responders, the third one represents the intruder and the last one the network.

Rewrite rules The rewrite rules describe the behavior of the honest agents involved in a session and the behavior of the intruder that tries to impersonate one of the agents. We will see that the invariants of the protocol are expressed by rewrite rules as well.

Each modification of the state of one of the participants to a session is described by a rewrite rule. At the beginning all the agents are in the state SLEEP waiting either to initiate a session or to receive a request for a new session.

When an initiator is in the state SLEEP, he initiates a session with one of the responders by sending the appropriate message as defined by the first step of the protocol. The following rewrite rule is used:

```
[initiator-1]
x+SLEEP+resp  || IN <> RE <> I <> lm                              =>
x+WAIT+N(x,y) || IN <> RE <> I <> x-->y:K(y)[N(x,y),N(di,di),A(x)]&lm
    where (Agent)y+std+init :=(extAgent) elemIA(RE)              end
```

In the above rewrite rule x and y are variables of type AgentId representing the identity of the initiator and the identity of the responder respectively. The initiator sends a nonce N(x,y) and his address (identity) encrypted with the public key of the responder and goes in the state WAIT where he waits for a response. Since only one nonce is necessary in this message, a dummy nonce N(di,di) is used in the second field of the message. The message is sent by including it in the multiset of messages available on the network.

Since the operator || is associative-commutative, when applying the rewrite rule initiator-1 the initiator x is selected non-deterministicly from the set of initiators. The identity of the responder y is selected non-deterministicly from the set of responders or from the set of intruders; in our case only one intruder. The non-deterministic selection of the responder is implemented by the strategy extAgent that selects at each application a new agent from the set given as argument.

If the destination of the previously sent message is a responder in the state SLEEP, then this agent gets the message and decrypts it if it is encrypted with his key. Afterwards, he sends the second message from the protocol to the initiator and goes in the state WAIT where he waits for the final acknowledgement:

```
[responder-1]
IN<> y+SLEEP+init  || RE <>I<> w-->y:K(y)[N(n1,n3),N(n2,n4),A(z)]&lm
=> IN<> y+WAIT+N(y,z) || RE <>I<> y-->z:K(z)[N(n1,n3),N(y,z),A(y)]&lm
```

One should notice that due to the associative-commutative definition of the operator & the position of the message in the network is not important. A non-associative-commutative definition would have implied several rewrite rules for describing the same behavior.

The condition that the message is encrypted with the public key of the responder is implicitly tested due to the matching that instantiates the variable y from y+SLEEP+init and K(y) with the same agent identity. Therefore, we do not have to add an explicit condition to the rewrite rule that remains simple and efficient.

Two other rewrite rules describe the other message exchanges from a session. When an initiator x and a responder y have reached the state COMMIT at the end of a correct session the nonce `N(y,x)` can be used as a symmetric encryption key for further communications between the two agents.

The intruder can be viewed as a normal agent that can not only participate to normal sessions but that tries also to break the security of the protocol by obtaining information that are supposed to be confidential. The network that serves as communication support is common to all the agents and therefore all the messages can be observed or intercepted and new messages can be inserted in it. There is no difficulty to implement the rules for the intruder in ELAN but for reasons of space they are omitted in this presentation.

The invariants of the protocol are easily represented by two rewrite rules describing the negation of the conditions that should be verified by the participants to the protocol session. If one of these two rewrite rules can be applied during the execution of the specification then the authenticity of the protocol is not ensured and an attack can be described from the trace of the execution.

Some additional properties on the multisets (of messages) can be expressed using unlabeled rewrite rules. For example the elimination of duplicates from a multiset of messages is represented by the rule

```
        [] m & m & l => m & l
```

that is applied implicitly after each application of any labeled rule.

Strategies The rewrite rules used to specify the behavior of the protocol and the invariants should be guided by a strategy describing their application. Basically, we want to apply repeatedly all the above rewrite rules in any order and in all the possible ways until one of the attack rules can be applied.

The strategy is easy to define in ELAN by using the non-deterministic choice operator `dk`, the `repeat*` operator representing the repeated application of a strategy and the ; operator representing the sequential application of two strategies:

```
[]attStrat  =>
  repeat*( dk(  attack-1, attack-2,
                intruder-1, intruder-2, intruder-3, intruder-4,
                initiator-1, initiator-2, responder-1, responder-2
  )); attackFound
```

The strategy tries to apply one of the rewrite rules given as argument to the `dk` operator starting with the rules for attacks and intruders and ending with the rules for the honest agents. If the application succeeds the state is modified accordingly and the `repeat*` strategy tries to apply a new rewrite rule on the result of the rewriting. When none of the rules is applicable, the `repeat*` operator returns the result of the last successful application. Since the `repeat*` strategy is sequentially composed with the `attackFound` strategy, this latter strategy is applied on the result of the `repeat*` strategy.

The strategy `attackFound` is nothing else but the rewrite rule:

```
    [attackFound]   ATTACK   =>   ATTACK              end
```

If an attack has not been found and therefore the strategy `attackFound` cannot be applied a backtrack is performed to the last rule applied successfully and another application of the respective rule is tried. If this is not possible the next rewrite rule is tried and if none of the rules can be applied a backtrack is performed to the previous successful application.

If the result of the strategy `repeat*` reveals an attack, then the `attackFound` strategy can be applied and the overall strategy succeeds. The trace of the attack can be recovered in the ELAN environment.

The trace obtained when executing the ELAN specification describes exactly the attack presented in [Low95] where the intruder impersonates an agent in order to establish a session with another agent.

The ELAN specification can be easily modified in order to reflect the correction shown sound in [Low96] and as expected, when the specification is executed with the modified rules no attacks are detected.

4 Conclusion

We have presented the ρ_T-calculus and we have seen that by making explicit the notion of rule, rule application and application result, the ρ_T-calculus allows us to describe in a simple yet very powerful manner the combination of algebraic and higher-order frameworks.

In the ρ_T-calculus the non-determinism is handled by using multisets of results and the rule application failure is represented by the empty set. Handling multisets is a delicate problem and the raw ρ_T-calculus, where the evaluation rules are not guided by a strategy, is not confluent but when an appropriate evaluation strategy is used the confluence is recovered.

The ρ_T-calculus is both conceptually simple as well as very expressive. This allows us to represent the terms and reductions from λ-calculus and conditional rewriting. Starting from this representation we showed how the ρ_T-calculus can be used to give a semantics to ELAN rules. This could be applied to many other frameworks, including rewrite based languages like ASF+SDF, ML, Maude or CafeOBJ but also production systems and non-deterministic transition systems.

We have shown how the ELAN language can be used as a logical framework for representing the Needham-Schroeder public-key protocol. This approach can be easily extended to other authentication protocols and an implementation of the TMN protocol has been already developed. The rules describing the protocol are naturally represented by conditional rewrite rules. The mixfix operators declared as associative-commutative allow us to express and handle easily the random selection of agents from a set of agents or of a message from a set of messages.

Among the topics of further research, let us mention the deepening of the relationship between the ρ_T-calculus and the rewriting logic [Mes92], the study of the models of the ρ_T-calculus, and also a better understanding of the relationship between the rewriting relation and the rewriting calculus.

References

[BKK$^+$96] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[CHL96] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cir99] H. Cirstea. Specifying authentication protocols using ELAN. In *Workshop on Modelling and Verification*, Besancon, France, December 1999.

[CK99a] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[CK99b] H. Cirstea and C. Kirchner. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, December 1999.

[Der85] N. Dershowitz. Computing with rewrite systems. *Information and Control*, 65(2/3):122–157, 1985.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[GKK$^+$87] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[Low95] G. Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public key protocol using CSP and FDR. In *Proceedings of 2nd TACAS Conf.*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166, Passau (Germany), 1996. Springer-Verlag.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Pro00] Protheo Team. The ELAN home page. WWW Page, 2000. `http://www.loria.fr/ELAN`.

[vdBvDK$^+$96] M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of asf+sdf. In M. Wirsing and M. Nivat, editors, *AMAST '96*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[Wol99] S. Wolfram. *The Mathematica Book*, chapter Patterns, Transformation Rules and Definitions. Cambridge University Press, 1999. ISBN 0-521-64314-7.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 68 - 77.

Objects in Test Tube Systems [*]

Erzsébet CSUHAJ-VARJÚ
and
György VASZIL

Computer and Automation Research Institute
Hungarian Academy of Sciences
Kende utca 13-17, 1111 Budapest, Hungary
E-mail: csuhaj/vaszil@sztaki.hu

Abstract

We introduce the notion of a test tube system with objects, a distributed parallel computing device operating with multisets of symbols, motivated by characteristics of biochemical processes. We prove that these constructs are suitable tools for computing, any recursively enumerable set can be identified by a TTO system. We also raise some open questions arising from the unconventional nature of this computational tool.

1 Introduction

Recently there have been a growing interest in investigating the principles and potentials for natural design and programmability in constructs simulating complex biomolecular systems. Among these investigations proposals for distributed parallel devices inspired by DNA-related structures or constructions motivated by biochemical processes are of particular interest.

Test tube systems based on splicing [1] or test tube systems with cutting and recombination operations [2] are examples for the first types of constructions. A test tube system is a finite collection of generative mechanisms (test tubes) which operate on strings (sets of strings or multisets of strings) using language theoretic operations motivated by the recombinant behaviour of DNA strands, and which communicate with each other by transferring the result of their computation. The notions were inspired by the famous experiment of L. M. Adleman computing an instance of the Hamiltonian path problem with DNA molecules in test tubes. Test tube systems realize universal computational devices, their computational capacity is equal to that of Turing machines ([1], [2]). For several other variants and related models the interested reader is referred to [5].

[*]Research supported in part by the Hungarian Scientific Research Fund "OTKA" Grant no. T 029615.

For computing devices of biochemical types, a recent paradigm, called P system was proposed in [6]. Since then the model has obtained increasing interest, for an early survey on the topic we refer to [7]. In these systems objects move among regions realizing cells of a membrane structure. Objects in the region can undergo operations which simulate biochemical processes. Since the same object can be present in a region in several copies, the model is based on multisets of objects, which makes the model to be closer to the realistic approximations of natural processes. Interesting questions are what can we say about the computational capacity of the different variants of these systems and related models, how to compute multisets, and how to measure complexity of these systems.

In this article we deal with these questions. We introduce and study so-called test tube systems with objects, which are models capturing certain characteristics of both test tube systems and P systems.

A test tube system with objects, a TTO system for short, is a finite collection of mechanisms operating with multisets of objects by performing operations called reactions among the objects and by communication which means the transfer of the contents of a test tube (the multiset of objects in the test tube) to another tube. The objects in the tubes are represented by symbols of an alphabet and a multiset of objects is given as a word over this alphabet with the same number of occurrences of a letter as the multiplicity of the object in the multiset which is identified by the letter. A reaction is prescribed by a rewriting rule of the form $u \to v$ where u and v are strings, u is not equal to the empty string. The meaning of this rule is that a multiset of objects represented by a word u is transformed to a multiset of objects represented by word v, supposing that the objects can form a chain (a structure) corresponding to u. If the obtained multiset is represented by ε, the empty word, then the objects forming the chain described by u disappear from the multiset. TTO systems compute multisets of objects by sequences of alternating steps: reaction and communication. Any computation starts from the initial configuration where each test tube contains a multiset of objects called its initial contents (this can be empty). The result of the computation is a set of multisets of objects that can be found at a dedicated tube, called the master, at any step of the computation starting from the initial configuration.

We prove that TTO systems are suitable tools for computation, any recursively enumerable set of integers can be obtained as the cardinality of the multiset that can be found at the master tube of a TTO system at some step of any computation starting from the initial configuration.

In addition to this result, we raise open questions arising from the nature of this unconventional computational tool.

2 Basic definitions

Throughout the paper we assume that the reader is familiar with formal language theory, for further details consult [4], [5], and [8].

The set of nonempty words over an alphabet Σ is denoted by Σ^+, if the empty word ε is included, then we use notation Σ^*. A set of strings $L \subseteq \Sigma^*$ is said to be a language over Σ. For a string $w \in L$, we denote the length of w by $lg(w)$ and for a set of symbols U, we denote by $|w|_U$ the number of occurrences of letters of U in w.

A multiset of objects M is a pair $M = (\Sigma, f)$, where Σ is an arbitrary (not necessarily finite) set of objects and f is a mapping $f : \Sigma \to N$; f assigns to each object in Σ its multiplicity in M. The set Σ is called the support of M.

If Σ is a finite set, then M is called a finite multiset.

The number of objects in a finite multiset of objects $M = (\Sigma, f)$, the cardinality of M, denoted by $card(M)$, is defined by $card(M) = \sum_{a \in \Sigma} f(a)$.

The reader can easily observe that any finite multiset of objects M with support $\Sigma = \{a_1, \ldots, a_n\}$ can be represented as a string w over alphabet Σ with $|w|_{a_i} = f(a_i)$, $1 \leq i \leq n$, the empty multiset is represented by $[\varepsilon]$. Clearly, all words obtained from w by permuting the letters can also represent M.

In the following we often will use this type of representation, and we will denote by $[w]$ the finite multiset of objects M with support Σ represented by word w over Σ.

Now we introduce the notion of a test tube system with objects and define how it functions. The notion captures certain features of test tube systems based on splicing and test tube systems based on cutting and recombination operations from DNA computing [1], [2] and P-systems [6].

Definition 2.1 A *test tube system with objects* (a TTO system, for short) is an $n+1$-tuple

$$\Gamma \quad = \quad (V, \Pi_1, \ldots, \Pi_n),$$

for $n \geq 1$, where

- V is a finite alphabet, the alphabet of objects in the system,
- $\Pi_i = (R_i, [w_i]),\ 1 \leq i \leq n$, is the i-th test tube, where
  - R_i is a finite set of rules $u \to v$, with $u \in V^+,\ v \in V^*$, the set of reaction rules in test tube Π_i,
  - $[w_i]$ is a multiset of objects from V represented by the word w, $w_i \in V^*$, called the initial contents (the axiom) of Π_i.

Test tube systems function through reactions processed in the tubes and communication which means the transfer of the contents of a tube to another tube. At any moment of time, the state of the test tube system is represented by the contents of the test tubes (the multiset of objects in the test tubes) at that moment.

Definition 2.2 Let $\Gamma = (V, \Pi_1, \ldots, \Pi_n),\ n \geq 1$, be a test tube system with objects. An n-tuple $([u_1], \ldots, [u_n])$ where $[u_i],\ 1 \leq i \leq n$, is a multiset of objects represented by string $u_i \in V^*$, is said to be a *configuration* (a *state*) of Γ. Multiset $[u_i]$, $1 \leq i \leq n$, is called the contents of the i-th test tube.

The initial configuration of Γ is $([w_1], \ldots, [w_n])$, where $[w_i]$ is the initial contents of test tube $\Pi_i,\ 1 \leq i \leq n$.

Now we define the reactions processed by the objects in test tube systems.

Definition 2.3 Let $\Gamma = (V, \Pi_1, \ldots, \Pi_n),\ n \geq 1$, be a test tube system with objects, and let $S_1 = ([u_1], \ldots, [u_n]),\ S_2 = ([v_1], \ldots, [v_n])$ be two configurations of Γ. We say that S_2 directly follows from S_1 by *reaction*, denoted by $S_1 \Longrightarrow_{rea} S_2$, if the following holds:

For each $i,\ 1 \leq i \leq n$, there are words $z_i \in V^*$, such that $[u_i] = [z_i]$ and $z_i \Longrightarrow v_i$ by applying R_i; that is, $z_i = \alpha_1 \ldots \alpha_m,\ v_i = \beta_1 \ldots \beta_m$ and for all $j,\ 1 \leq j \leq m$, $\alpha_j \to \beta_j \in R_i$.

Notice, that $z_i,\ 1 \leq i \leq n$, can differ from u_i. Reaction rules $R_i,\ 1 \leq i \leq n$, prescribe possible reactions between objects which are poured into the test tube separately but they can form structures in the tube. A reaction is successful if and only if the objects in the tube can form a chain which corresponds to a string which can be rewritten by parallel application of some rewriting rules from the set of reaction rules of the tube. The computation process gets blocked if it is not possible to find a representation where all symbols of the representing string are rewritten; that is, all objects participate in the reaction. If the rules have only one symbol on their left-hand side, then there is no interaction between the objects in the test tube, the chosen representation makes no difference in the result of the reaction.

It is an interesting question, how many string representations of the multiset of objects in the tube can be found to induce a successful reaction in the tube. That is, how many strings composed from the letters representing the objects can be rewritten by parallel application of the given rewriting rules. This property, called *reaction capacity*, expresses determinism. Clearly, if the reaction rules are context-free rules all strings are "good" strings according to this property. It would be interesting to study reaction capacity of test tubes according to different presentations of the reaction rule set.

Notice also, that there are several possibilities to define reactions. Here we require that all objects in the tube must participate in the reaction, that is, all objects must appear on the left-hand side of a rule. Other possibilities would be to allow some objects (substrings in the representation) not to be rewritten at all, or to require that in each reaction not necessarily all, but the maximal possible number of objects must participate. Again, it would be interesting to study the question how we can minimize the number of objects in the test tubes not taking part in any reaction and, whether there are different presentations of reaction rules which imply the same multisets of objects not involved in any reaction.

These properties are *size complexity measures* for test tubes with objects.

After the reactions, the contents of the tubes are redistributed by communication.

Definition 2.4 Let $\Gamma = (V, \Pi_1, \ldots, \Pi_n),\ n \geq 1$, be a test tube system with objects, and let $S_1 = ([u_1], \ldots, [u_n]),\ S_2 = ([v_1], \ldots, [v_n])$ be two configurations of Γ.

We say that S_2 directly follows from S_1 by *communication*, denoted by $S_1 \Longrightarrow_{com} S_2$, if the following condition holds:

There exists a set of ordered pairs of test tubes $C \subseteq \{(\Pi_i, \Pi_j) \mid 1 \leq i, j \leq n\}$ with the property that if $(\Pi_i, \Pi_j) \in C$ and $(\Pi_i, \Pi_k) \in C$, then $j = k$ and for each $i,\ 1 \leq i \leq n$,

- either $[v_i] = [u_i u_{i_1} \dots u_{i_s}]$, where $(\Pi_{i_j}, \Pi_i) \in C,\ 1 \leq j \leq s,\ s \leq n$, and there is no $k,\ 1 \leq k \leq n$, with $(\Pi_i, \Pi_k) \in C$, or
- if for some $k,\ 1 \leq k \leq n,\ (\Pi_i, \Pi_k) \in C$, then $[v_i] = [u_{i_1} \dots u_{i_s}],\ (\Pi_{i_j}, \Pi_i) \in C$ $1 \leq j \leq n,\ s \leq n$.

We call C the actual communication graph in this communication step.

Communication in a configuration S is realized by redistributing the contents of the test tubes, pouring the contents of a tube Π_i into another tube $\Pi_j,\ 1 \leq i, j, \leq n$, if the ordered pair (Π_i, Π_j) is an element of C. The pairs are chosen before the communication in such a way that the contents of each test tube is transferred to at most one other tube.

If for some $i, j,\ 1 \leq i, j \leq n$, the contents $[\alpha_i]$ of the tube Π_i is poured into Π_j having contents $[\alpha_j]$, then the objects of the two test tubes are mixed, we obtain the new test tube contents $[\alpha_i \alpha_j]$ in Π_j.

The reader can invent several other ways of communication. For example, in the case of test tube systems based on splicing and that of with cutting and recombination operations the contents of the test tube to be transferred was allowed to be amplified, the same contents could be transferred in several copies to different test tubes. We also can prescribe fixed or dynamically changing graphs for the communicating test tubes, or we can control communication through filters (multisets of objects prescribed to be included in the communicated contents). The reader can find several examples for these types of constructs in the literature [5].

The sequence of reactions and contents redistributions (communications) define a computation in Γ.

Definition 2.5 A *computation* in a TTO system $\Gamma = (V, \Pi_1, \dots, \Pi_n)$ is a sequence of states, $S_j,\ j \geq 0$, such that

- $S_j \Longrightarrow_{rea} S_{j+1}$ for $j = 2k,\ k \geq 0$, and
- $S_j \Longrightarrow_{com} S_{j+1}$ for $j = 2k + 1,\ k \geq 0$.

Let also $\Longrightarrow$ denote a computation step, either $\Longrightarrow_{rea}$ or $\Longrightarrow_{com}$, and let $\Longrightarrow^*$ denote the reflexive and transitive closure of $\Longrightarrow$.

The result of a computation in a TTO system is the set of multisets of objects which can be found at a given node of the system (the master) after processing the reactions during the computation.

Definition 2.6 Let $\Gamma = (V, \Pi_1, \dots, \Pi_n),\ n \geq 1$, be a TTO system. The *computational capacity* of Γ is the set of multisets

$$L(\Gamma) \quad = \quad \{[\beta_1] \mid ([w_1], \dots, [w_n]) \Longrightarrow^* ([\alpha_1], \dots, [\alpha_n]) \Longrightarrow_{rea} ([\beta_1], \dots, [\beta_n])\},$$

where component Π_1 is the *master* and $([w_1], \dots, [w_n])$ is the initial state of Γ.

We give an example for a TTO system.

Example 1 *Let $G = (V, P_1, \ldots, P_n, w)$, $n \geq 1$, be a TOL system, a tabled interactionless Lindenmayer system. (These systems are parallel language generating mechanisms for modelling developmental systems.) In a TOL system $G = (V, P_1, \ldots, P_n, w)$, $n \geq 1$, V denotes the alphabet of the system, and $w \in V^*$ is the axiom. P_i, $1 \leq i \leq n$, are sets of pure context-free rules over V, called tables, such that each P_i contains at least one production for each letter in V. A direct derivation step in G is defined as $a_1 \ldots a_m \Longrightarrow u_1 \ldots u_m$, $m \geq 1$, where $a_i \in V$, $u_i \in V^*$, $1 \leq i \leq m$, and $a_i \to u_i$, $1 \leq i \leq m$, is in P_j for some j, $1 \leq j \leq n$. Thus, each letter in the word is rewritten by applying a corresponding rule of a table. In a derivation step only one of the tables can be used. The reader can easily observe that G can be interpreted as a TTO system $\Gamma = (V, \Pi_1, \ldots, \Pi_n)$: V denotes the alphabet of the objects, P_i corresponds to the the set of reaction rules of the i-th test tube, and w is a word representing the initial contents of the first test tube, the initial contents of the other tubes are empty. Any derivation step in G by using a table P_i, $1 \leq i \leq n$, corresponds to a reaction in test tube Π_i, and the change of a table corresponds to a communication in the TTO system Γ. Moreover, the Parikh vector of a word obtained from the axiom of G by a derivation d using table P_j, $1 \leq j \leq n$, at the last step corresponds to the contents of test tube Π_j obtained by a computation in Γ simulating derivation steps of d. (For a word w over an alphabet $V = \{a_1, \ldots, a_n\}$, $n \geq 1$, the n-tuple of integers $(|w|_{a_1}, \ldots |w|_{a_n})$ is called its Parikh vector, that is, the values of the Parikh vector give the multiplicity of the occurrences of the different letters in the word.)*

3 Computing by TTO systems

In the following we shall demonstrate that recursively enumerable sets can be computed by TTO systems, that is, for any recursively enumerable language L we can construct a TTO system such that the cardinality of any multiset resulted by any computation in the TTO system is equal to an integer that represents a word of the language in a unique manner.

First, we need a technical result.

It is obvious that any recursively enumerable language over an alphabet $\Sigma = \{a_1, \ldots, a_n\}$, $n \geq 1$, determines a recursively enumerable set of integers since any word $w \in \Sigma^*$ can be considered as a number written in $(n+1)$-ary notation, where each symbol a_i, $1 \leq i \leq n$, represents the digit i. This way each different string corresponds to a different integer, and the notation of these integers do not contain the digit 0, so we do not have to consider strings corresponding to numbers with leading zeros. This means that the value of such a representing integer uniquely determines the string it represents. In the following, for a word $w \in \Sigma^*$ we denote by $val(w)$ the representing integer.

Our result will be based on the simulation of the so-called Extended Post Correspondence by TTO systems.

Definition 3.1 Let $\Sigma = \{a_1, \ldots, a_n\}$, $1 \leq n$, be an alphabet. An *Extended Post*

Correspondence (an EPC) is a pair

$$P = (\{(u_1, v_1), \ldots, (u_m, v_m)\}, (z_{a_1}, \ldots, z_{a_n})),$$

where $u_i, v_i, z_{a_j} \in \{0, 1\}^*,\ 1 \le i \le m,\ 1 \le j \le n$. The *language represented by P*, denoted by $L(P)$ is the following:

$$L(P) = \{x_1 \ldots x_r \in \Sigma^* \mid \text{there are } i_1, \ldots i_s \in \{1, \ldots, m\}, s \ge 1, \text{ such that } v_{i_1} \ldots v_{i_s} = u_{i_1} \ldots u_{i_s} z_{x_1} \ldots z_{x_r}\}.$$

It is known (see [3]) that for each recursively enumerable language L there exists an EPC system P such that $L(P) = L$. Clearly, the statement remains true if words $u_i, v_i, z_{a_j},\ \ 1 \le i \le m,\ 1 \le j \le n$ are defined over alphabet $\{1, 2\}$. Let us use this modified version of the EPC. According to the above theorem, if we consider an EPC system P, then a word $w = x_1 \ldots x_r \in \Sigma^*$ is in L if and only if there are indices $i_1, \ldots, i_s, \in \{1, \ldots, m\},\ s \ge 1$, such that the two numbers $v_{i_1} \ldots v_{i_s}$ and $u_{i_1} \ldots u_{i_s} z_{x_1} \ldots z_{x_r}$ with digits from $\{1, 2\}$ are equal.

Thus, we can check if a string $w = x_1 \ldots x_r$ is an element of the language $L(P)$ in the following manner: We start from a string $u_{i_1} v_{i_1}$ and then append strings from $\{u_1 \ldots, u_m\}$ to u_{i_1} and strings from $\{v_1, \ldots, v_m\}$ to v_{i_1}. At the end of the procedure, we obtain a string of the form $u_{i_1} \ldots u_{i_s} v_{i_1} \ldots v_{i_s}$. Then, we continue by appending strings from $\{z_{a_1}, \ldots, z_{a_n}\}$ to $v_{i_1} \ldots v_{i_s}$, obtaining $u_{i_1} \ldots u_{i_s} v_{i_1} \ldots v_{i_s} z_{x_1} \ldots z_{x_r},\ x_i \in \Sigma,\ 1 \le i \le r$. Finally, we check whether or not the two words $\alpha = u_{i_1} \ldots u_{i_s}$ and $\beta = v_{i_1} \ldots v_{i_s} z_{x_1} \ldots z_{x_r}$ have the same value as numbers.

Our idea is based on the above considerations. We generate a multiset representing the word $w = x_1 \ldots x_k \ldots x_r \in \Sigma^*$ by computations in test tubes as follows. At any moment of time, the string $x_1 \ldots x_k \alpha\beta$, where $\alpha = u_{i_1} \ldots u_{i_l}$ and $\beta = v_{i_1} \ldots v_{i_l} z_{x_1} \ldots z_{x_k}$, is present in a test tube represented as a multiset including objects A, B, C, where the multiplicities of A and B are equal to the value of α and β as numbers with digits from $\{1, 2\}$, and the multiplicity of C is equal to the value of $x_1 \ldots x_k$ as an $(n + 1)$-ary number when a symbol $a_i,\ 1 \le i \le n$, from Σ is interpreted as the $(n + 1)$-ary digit i.

When we pour the contents of a tube representing the string $x_1 \ldots x_k \alpha\beta$ into another one, a reaction takes place changing the number of objects A, B, and C to simulate the appending of a pair (u_i, v_i) or $(x, z_x),\ 1 \le i \le m,\ x \in \Sigma$, to α and β, or to $x_1 \ldots x_k$ and β, respectively. Then, the obtained multiset is poured into another tube again. After repeating these steps several times, the multiset of objects is poured into a tube dedicated for deciding whether the objects A and B have the same multiplicity. This is done by simple reactions, namely applying rules $AB \to \varepsilon$. These reactions check whether the computed words α and β have the same value as numbers. After a successful reaction, the tube will contain the object C in as many occurrences as the $(n + 1)$-ary value corresponding to the string $x_1 \ldots x_r$.

Theorem 3.1 *For any recursively enumerable language L we can construct a TT0 system Γ such that*

$$\{card(M) \mid M \in L(\Gamma)\} = \{val(w) \mid w \in L\}.$$

Proof. Let L be a recursively enumerable language over an alphabet $\Sigma = \{a_1, \ldots, a_t\}$ and let P be an Extended Post Correspondence with $L = L(P)$. Without loss of generality we may assume that P is in the slightly modified form given above.

We construct a TTO system Γ with the property that $[w] \in L(\Gamma)$ if and only if $card([w]) = val(u)$ for some u in L, and reversely, for any $u \in L$ there is a multiset $[w]$ in $L(\Gamma)$ such that $card[w] = val(u)$. Let

$$P \;=\; (\{(u_1, v_1), \ldots, (u_m, v_m)\}, (z_1, \ldots, z_t)),$$

where $u_i, v_i, z_j \in \{1, 2\}^*,\ 1 \leq i \leq m,\ 1 \leq j \leq t$, and let

$$\Gamma \;=\; (V, \Pi_0, \Pi_1, \ldots, \Pi_n, \Pi_{n+1}),$$

where $\Pi_i = (R_i, [w_i]),\ 0 \leq i \leq n+1$, with $n = m + t$, the master is Π_{n+1}, and

$$V \;=\; \{\$, \#, \&, A, B, C\}.$$

Let

$$\begin{aligned} [w_0] &= [\$], \\ R_0 &= \{\$ \to \$\}. \end{aligned}$$

Now for $1 \leq i \leq m$, let

$$\begin{aligned} [w_i] &= [\varepsilon], \\ R_i &= \{\$ \to A^{val(u_i)}\#B^{val(v_i)}, \# \to A^{val(u_i)}\#B^{val(v_i)}, \\ &\qquad A \to A^k, B \to B^l \mid k = 3^{lg(u_i)},\ l = 3^{lg(v_i)}\}. \end{aligned}$$

For $1 \leq i \leq t$, let

$$\begin{aligned} [w_{m+i}] &= [\varepsilon], \\ R_{m+i} &= \{\# \to C^{val(a_i)}\&B^{val(z_i)}, \& \to C^{val(a_i)}\&B^{val(z_i)}, C \to C^{t+1}, B \to B^l \mid \\ &\qquad l = 3^{lg(z_i)}\}, \end{aligned}$$

and let also for a fixed $w \in L$

$$\begin{aligned} [w_{n+1}] &= [C^{val(w)}], \\ R_{n+1} &= \{\#C^{val(w)} \to \varepsilon, \&C^{val(w)} \to \varepsilon, AB \to \varepsilon, C \to C\}. \end{aligned}$$

This system simulates the Extended Post Correspondence P as outlined above. The test tubes can be grouped into four types according to their function: the initial tube Π_0, tubes of the second type $\Pi_i,\ 1 \leq i \leq m$, tubes of the third type $\Pi_i,\ m+1 \leq i \leq n$, and the master tube Π_{n+1}.

In the initial state, the tubes are empty, except Π_0, where the reaction leaves the object \$ unchanged, and the master, where a multiset representing a word w of L is present. To start the computation, we can pour the contents of the initial tube into a tube of the second type, $\Pi_i,\ 1 \leq i \leq m$, where the reactions change the object \$ to # and add several As and Bs corresponding to the value of u_i and v_i. Now we can repeat the procedure several times by leaving the contents in the tube or pouring it into another tube of the second type, creating this way a multiset representing a string $u_1 \dots u_k v_1 \dots v_k$ by $[A^{val(u_1 \dots u_k)} \# B^{val(v_1 \dots v_k)}]$. (We note that according to the given way of communication the multiset can remain in the tube after the reaction.)

If we consider a string $\alpha\beta$, where $\alpha \in \{u_1, \dots, u_m\}^*$, $\beta \in \{v_1, \dots, v_m\}^*$, and a representation of this string $[A^{val(\alpha)} \# B^{val(\beta)}]$, we can get the representation of $\alpha u_j \beta v_j$ by pouring the objects above into test tube Π_j. In this tube the number of A and B objects are multiplied by $3^{lg(u_j)}$ and $3^{lg(v_j)}$ respectively, and then $val(u_j)$ As and $val(v_j)$ Bs are added. This way we obtain $[A^{val(\alpha u_j)} \# B^{val(\beta v_j)}]$, the multiset representing $\alpha u_j \beta v_j$.

After the tubes of the second type, the tubes of the third type, Π_i, $m+1 \leq i \leq n$, or the master tube can be used. If the master tube is used and the reaction is successful, then the multiplicities of A and B objects are equal. This means that we have a representation of string $u_1 \dots u_r v_1 \dots v_r$ where $u_1 \dots u_r = v_1 \dots v_r$, thus the empty string, ε, is part of the language represented by P, the empty multiset, the representation of ε, is computed by Γ.

If a multiset of objects $[A^{val(\alpha)} \# B^{val(\beta)}]$ is poured into a tube of the third type, Π_{m+j}, then the object # is changed to & and a number of B and C objects are added, so the obtained multiset corresponds to the string $\alpha x_j \beta z_{x_j}$, for some $x_j \in \Sigma$. It is done in the same way as above by multiplying and adding, obtaining the multiset $[A^{val(\alpha)} C^{val(x_j)} \& B^{val(\beta z_{x_j})}]$. The tubes of the third type can be again used repeatedly, and then we have a multiset $[A^{val(\alpha)} C^{val(w)} \& B^{val(\beta\gamma)}]$ with $\alpha \in \{u_1, \dots, u_m\}^*$, $\beta \in \{v_1, \dots, v_m\}^*$, and $\gamma = z_{x_{i_1}} \dots z_{x_{i_s}}$, where $w = x_{i_1} \dots x_{i_s}$.

If we pour this into the master tube, Π_{n+1}, then a successful reaction means that number of occurrences of As and Bs are the same; that is $\alpha = \beta\gamma$, and $w \in L$, α, β, γ, w as above, and the tube contains C objects in as many occurrences as the $(t+1)$-ary value of $w = x_{i_1} \dots x_{i_s}$. If the reaction is not successful, $w \notin L$, then the work of the system is blocked. By the construction of Γ, no multiset can be computed with cardinality being different from the value of an integer representation of some word in L. $\square$

4 Final remarks

The unconventional nature of TTO systems leads to several interesting problems. For example, it would be interesting to study how economical is a TTO system, that is, how economically the reactions are processed in the whole system, how many test tubes are necessary to obtain the same result of computation, how freely we can choose the reaction rules. Another topic is the way of communication, how can it be organized to obtain the same or different result. Clearly, many questions and open problems remain for further investigations.

References

[1] E. Csuhaj-Varjú, L. Kari, Gh. Păun, Test Tube Distributed Systems Based on Splicing. Computers and Artificial Intelligence 15(2) (1996), 21-232.

[2] R. Freund, E. Csuhaj-Varjú, F. Wachtler, Test Tube Systems with Cutting/Recombination Operations. In: Proc. Pacific Symp. on BIOCOMPUTING'97. ED. by R.B. Altman et al., World Scientific, Singapore, 1997, 163-175.

[3] V. Geffert, Context-free-like forms for phrase structure grammars. Proc. MFCS'88, LNCS 324, Springer Verlag, 1988, 309-317.

[4] Handbook of Formal Languages. Vol. I-III. Eds. by G. Rozenberg, A. Salomaa, Springer-Verlag, Heidelberg, 1997.

[5] Gh. Păun, G. Rozenberg, A. Salomaa, DNA-Computing: New Computing Paradigms. Springer Verlag, Heidelberg, 1998.

[6] Gh. Păun, Computing with membranes. J. of Computer and Systems Sciences, to appear. (Also in: TUCS Research Report No. 208, November 1998.)

[7] Gh. Păun, Computing with membranes. An introduction. Bulletin of the EATCS 67 (1999), 139-152.

[8] G. Rozenberg, A. Salomaa, The Mathematical Theory of L Systems. Academic Press, New York, 1981.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 78 - 99.

A uniform approach to constraint-solving for lists, multisets, compact lists, and sets

Agostino Dovier[*] Carla Piazza[†] Gianfranco Rossi[‡]

Abstract

Lists, multisets, and sets are well-known data structures whose usefulness is widely recognized in various areas of Computer Science. These data structures have been analyzed from an axiomatic point of view with a parametric approach in [12] and the relevant unification algorithms have been also parametrically developed. In this paper we extend these results considering more general constraints including not only equality but also membership constraints as well as their negative counterparts. This amounts to define the privileged structures for the considered axiomatic theories and to solve the relevant constraint satisfaction problems in each of the theories. Like in [12], moreover, we adopt a highly parametric approach which allows all the results obtained separately for each single theory to be easily combined so as to obtain a general framework where it is possible to deal with more than one data structure at a time. **Keywords:** Constraints, Computable Set and Multiset Theory.

1 Introduction

Programming and specification languages allow the user to specify aggregation of elementary data objects and, in turn, aggregation of aggregates. Besides the well-known example of arrays, also lists, multisets, and sets are other important forms of data aggregates whose usefulness is widely recognized in various areas of Computer Science. Lists are the "classical" example used to introduce dynamic data structures in imperative programming languages. They are the fundamental data structure in the functional language LISP, and list predicates, such as *member* and *append*, are among the first predicates that are taught to students of the logic programming language PROLOG.

Sets are the main data structure used in specification languages (e.g., in Z [25]) and in high-level declarative programming languages [5, 13, 18, 16]; but also imperative programming languages can take advantage from the set data abstraction (e.g., SETL [26]).

Multisets emerge as the most natural data structure in several interesting applications. Solutions to the equation $x^4 - 2x^2 + 1 = 0$ are better described by the multiset $\{\!\{ -1, -1, 1, 1 \}\!\}$ rather than by the set $\{-1, -1, 1, 1\}$ which is equivalent to $\{-1, 1\}$. As explained in [28], *sets came to mean types of objects, while multisets*

[*]Dip. Scientifico-Tecnologico, Univ. di Verona. Strada Le Grazie 15, 37134 Verona (Italy). dovier@sci.univr.it

[†]Dip. di Matematica e Informatica, Univ. di Udine. Via Le Scienze 206, 33100 Udine (Italy). piazza@dimi.uniud.it

[‡]Dip. di Matematica, Univ. di Parma. Via M. D'Azeglio 85/A, 43100 Parma (Italy). gianfr@prmat.math.unipr.it

are based on tokens. This justifies the use of multisets in describing processes which consume resources. In particular, multisets over some set of basic elements (urelements) can be perfectly connected to fragments of linear logic [28]. Multisets are the fundamental data structure of the Gamma coordination language [3], based on the chemical metaphor, and of the Chemical Abstract Machine [4]: a multiset can be seen as a *solution* containing molecules that can react inside it. Using this metaphor it is natural to write parallel algorithms. For instance, assume that a multiset contains all the numbers between 2 and n and consider the multiset rewriting rule 'x destroys one of its multiples'. Several process can run in parallel inside the multiset applying the rule; at the end of the execution, only the prime numbers from 2 to n remain in the multiset [3]. Some issues on the relevance of multisets in Databases and the related complexity problems can be found in [17].

The basic difference between lists, multisets, and sets lies in the importance of order and/or repetitions of their elements: in lists both order and repetitions of elements are important; in multisets the order is immaterial, whereas the repetitions are important; in sets order and repetitions are not taken into account.

These three data structures have been analyzed from an axiomatic point of view in [12]. The axiomatizations provided in that paper induce a lattice of four points, having sets as top and lists as bottom, as shown in the figure below. In this lattice,

$$\begin{tikzcd}
 & \text{Sets} & \\
\text{Multisets} \arrow[ur] & & \text{Compact lists} \arrow[ul] \\
 & \text{Lists} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

between sets and lists, we find both multisets and a new data structure, called *compact lists.* Compact lists are lists in which contiguous occurrences of the same element are immaterial: a property complementary to that characterizing multisets. Their practical usage in programming has not been explored yet, although some possible examples are suggested in [12].

Lists, multisets, compact lists and sets have been studied in the context of (Constraint) Logic Programming (CLP) languages. In this context all these data structures are conveniently represented as terms, using four different data aggregate constructors endowed with the proper interpretations. The theories studied are *hybrid*, i.e. they can deal with interpreted function symbols as well as with an arbitrary number of free constant and function symbols (technically, we are in a *general* context). [12], however, focuses only on *equality* between terms in each of the four theories. This amounts to solve the relevant problems of unification in the equational theories describing the properties of the four considered data structures. Unification algorithms for these four data structures are provided in [12]; NP-unification algorithms for sets and/or multisets are also presented in [1, 10].

In this paper we extend the results of [12] to the case of more general constraints. The constraints we consider are arbitrary conjunctions of literals (i.e., positive and negative atoms) based on both equality and membership predicate symbols. The problem of dealing with such kind of constraints in the context of CLP languages has been already faced in [14], but limited to the set data structure. In this paper, in contrast, we face the same problem for all the four data structures mentioned above. We identify the privileged models for the axiomatic theories used to describe the considered data structures. We define a notion of (satisfiable) solved form for constraints that are conjunctions of positive and negative equality and membership constraints. We develop the rewriting algorithms which map these constraints into solved form constraints—proved to be correct and terminating—for all the four theories.

The whole presentation will be parametric with respect to the considered axiomatic theories, high-lightening differences and similarities between the four aggregates. As a consequence, the proposed solutions (axiomatic theories, structures, constraint satisfiability procedures) can be easily combined so as to account for more than one data structure at a time.

The paper is organized as follows. In Section 2 we fix the overall notation and we recall from [12] the (parametric) axiomatic presentation of the first-order theories we deal with. In Section 3 we define the privileged models and we show that they correspond with the related theories. We also present a global notion of solved form for constraints that ensures satisfiability in the four models. In Section 4 we briefly discuss constraint solving when constraints are conjunctions of equality atoms (unification problem) and we recall the results from [12]. In Section 5 we describe, for each kind of data structure, the constraint rewriting procedures used to eliminate the literals not in solved form possibly occurring in a given constraint, while in the next section, Section 6, we show how to solve parametrically the general satisfiability problem for the admissible constraints. In Section 7 we show how it is possible (and simple) to combine the procedures developed in order to obtain a unique general framework. Finally, some conclusions are drawn in Section 8. Due to lack of space, we omit all the proofs. They will be available in a forthcoming technical report of the University of Parma toghether with the analysis of Bag theories with multi-membership.

2 Preliminaries

We assume basic knowledge of first-order logic (e.g., [15, 7]). A first-order language $\mathcal{L} = \langle \Sigma, \mathcal{V} \rangle$ is defined by a *signature* $\Sigma = \langle \mathcal{F}, \Pi \rangle$ composed by a set $\mathcal{F}$ of constant and function symbols and a set Π of predicate symbols, and by a denumerable set $\mathcal{V}$ of logical variables.

Usually, capital letters X, Y, Z, etc. will be used to represent variables, f, g, etc. to represent function symbols, and p, q, etc. to represent predicate symbols. We will use $\bar{X}$ to denote a (possibly empty) sequence of variables. $T(\mathcal{F}, \mathcal{V})$ ($T(\mathcal{F})$) denotes the set of first-order terms (resp., ground terms) built from $\mathcal{F}$ and $\mathcal{V}$ (resp., $\mathcal{F}$). Given a sequence of terms $t_1, \dots, t_n$, $FV(t_1, \dots, t_n)$ will be used to denote the set of all the variables which occur in at least one of the terms t_i. When the context is clear, we will use $\bar{t}$ to denote a sequence $t_1, \dots, t_n$ of terms.

An *atomic formula* (atom) is an object of the form $p(t_1, \dots, t_n)$, where $p \in \Pi$, $ar(p) = n$ and $t_i \in T(\mathcal{F}, \mathcal{V})$. The *formulae* are built up from the atomic ones using first-order connectives $(\wedge, \vee, \neg, \dots)$ and quantifiers $(\exists, \forall)$. We assume the standard notion of *free* variables and we use $FV(\varphi)$ to denote the set of free variables in the first-order formula φ. If $FV(\varphi) = \emptyset$, then the formula is said to be *closed.* $\vec{\exists}\varphi$ ($\vec{\forall}\varphi$) denotes the existential (universal) closure of the formula φ, namely $\exists X_1 \cdots X_n\, \varphi$ ($\forall X_1 \cdots X_n\, \varphi$), where $\{X_1, \ldots, X_n\} = FV(\varphi)$.

A Σ-*structure* (or, simply, a *structure*) $\mathcal{A}$ is composed by a non-empty domain A and by an interpretation function $(\cdot)^{\mathcal{A}}$ which assigns functions and relations on A to the symbols of Σ. A *valuation* σ is a function from a subset of $\mathcal{V}$ in A. Each valuation can be exteded to a function from $T(\mathcal{F}, \mathcal{V})$ in A and to a function from the set of formulae over $\mathcal{L}$ on the set $\{\mathtt{false}, \mathtt{true}\}$. A valuation σ is said a *successful valuation* of φ if $\sigma(\varphi) = \mathtt{true}$.

A *(first-order) theory* $\mathcal{T}$ on $\mathcal{L}$ is a set of closed first-order formulae of $\mathcal{L}$, such that each closed formula of $\mathcal{L}$ which can be deduced from $\mathcal{T}$ is in $\mathcal{T}$. A *(first-order) set of axioms* Ax on $\mathcal{L}$ is a set of closed first-order formulae of $\mathcal{L}$. A set of axioms Ax is said to be the an *axiomatization* of $\mathcal{T}$ if $\mathcal{T}$ is the smallest theory such that $Ax \subseteq \mathcal{T}$. Sometimes we use the term *theory* also to refer to an axiomatization of the theory.

A *substitution* is a mapping $\theta : \mathcal{V} \longrightarrow T(\mathcal{F}, \mathcal{V})$. A substitution is then extended inductively to terms as usual. With ε we denote the empty substitution, namely the substitution such that $\varepsilon(x) = x$ for all variables x. A substitution θ is a $\mathcal{T}$-*unifier* of two terms t, t' if $\mathcal{T} \models \vec{\forall}(\theta(t) = \theta(t'))$ [27].

A *constraint (admissible constraint)* is a conjunction of *literals*, namely atomic formulae or negation of atomic formulae. When C is a constraint, $|C|$ is used to denote the number of occurrences of variables, constant, function, and predicate symbols in C.

Given a theory $\mathcal{T}$ on $\mathcal{L}$ and a structure $\mathcal{A}$, $\mathcal{T}$ and $\mathcal{A}$ *correspond* on the set of admissible constraints Adm [19] if, for each constraint $C \in Adm$, we have that $\mathcal{T} \models \vec{\exists}(C)$ if and only if $\mathcal{A} \models \vec{\exists}(C)$. This property guarantees that $\mathcal{A}$ is a canonical model of $\mathcal{T}$ with respect to Adm: if C is an element of Adm and we know that C is satisfiable in $\mathcal{A}$ then it will be satisfiable in all the models of $\mathcal{T}$.

The following binary function symbols are introduced to denote lists, multisets, compact lists, and sets:

$[\cdot\,|\,\cdot]$	for lists,	$\{\!\!\{\cdot\,|\,\cdot\}\!\!\}$	for multisets,
$[\![\cdot\,|\,\cdot]\!]$	for compact lists,	$\{\cdot\,|\,\cdot\}$	for sets.

The empty list, multiset, compact list, and set are all denoted by the constant symbol `nil`. We use simple syntactic conventions and notations for terms built using these symbols. In particular, the list $[\,s_1\,|\,[\,s_2\,|\,\cdots\,[\,s_n\,|\,t\,]\cdots]]$ will be denoted by $[s_1, \ldots, s_n\,|\,t]$ or simply by $[s_1, \ldots, s_n]$ when t is `nil`. The conventions used for lists will be exploited also for multisets, compact lists, and sets.

In [12] it is proposed a uniform parametric axiomatization of the data structures lists, multisets, compact lists, and sets that we briefly recall below. In each axiomatic theory $\mathcal{T}$ used to describe these data structures we have that $\Pi_{\mathcal{T}} = \{=, \in\}$ and $\mathcal{F}_{\mathcal{T}}$ contains the constant symbol `nil`, exactly one among $[\cdot\,|\,\cdot]$, $\{\!\!\{\cdot\,|\,\cdot\}\!\!\}$, $[\![\cdot\,|\,\cdot]\!]$, or $\{\cdot\,|\,\cdot\}$, plus possibly other (free) constant and function symbols.

These theories, therefore, are *hybrid theories*: the objects they deal with are built out of interpreted as well as uninterpreted symbols. In particular, lists (multisets, compact lists, sets) may contain uninterpreted Herbrand terms as well as other lists (resp., multisets, compact lists, sets). Moreover, all the data aggregates can be built by starting from any ground uninterpreted Herbrand term—called the *kernel* of the data structure—and then adding to this term the other elements that compose the aggregate. We refer to this kind of data structures as *colored* hybrid data structures (namely, lists, multisets, compact lists, and sets).

2.1 Lists

Let us consider a first-order language $\mathcal{L}_{List} = \langle \Sigma_{List}, \mathcal{V} \rangle$ over a signature $\Sigma_{List} = \langle \mathcal{F}_{List}, \Pi \rangle$ such that the binary function symbol $[\cdot\,|\,\cdot]$ and the constant symbol `nil` are in $\mathcal{F}_{List}$, and $\Pi = \{=, \in\}$. A first-order theory for lists over the language $\mathcal{L}_{List}$—called *List*—is shown in the figure below:

(K)	$\forall x\, y_1 \cdots y_n$	$(x \notin f(y_1, \dots, y_n))$	$f \in \mathcal{F}_{List}, f \not\equiv [\,\cdot\,|\,\cdot\,]$
(W)	$\forall y\, v\, x$	$(x \in [\,y\,|\,v\,] \leftrightarrow x \in v \vee x = y)$	
(F_1)	$\forall x_1 \cdots x_n y_1 \cdots y_n$	$\left(\begin{array}{c} f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \\ \to x_1 = y_1 \wedge \cdots \wedge x_n = y_n \end{array}\right)$	$f \in \mathcal{F}_{List}$
(F_2)	$\forall x_1 \cdots x_m y_1 \cdots y_n$	$f(x_1, \dots, x_m) \neq g(y_1, \dots, y_m)$	$f, g \in \mathcal{F}_{List}, f \not\equiv g$
(F_3)	$\forall x$	$(x \neq t[x])$	
	where $t[x]$ *denotes a term, having* x *as proper subterm*		

The three axiom schemata (F_1), (F_2), and (F_3) (called freeness axioms, or Clark's equality axioms—see [8]) have been originally introduced by Mal'cev in [22]. Observe that axiom (F_1) holds for $[\,\cdot\,|\,\cdot\,]$ as a particular case. Axiom (F_3) states that there does not exist a term which is also a subterm of itself. In particular if $x = [x]$ had solutions, then, by (W), $x \in x$ would also have solutions. Thus, axiom schema (F_3) is a weak form of the *foundation axiom* (see, e.g., [21]) which has the aim, among others, of guaranteeing the *acyclicity* of membership. Note that (K) implies that $\forall x\, (x \notin \texttt{nil})$.

2.2 Multisets

Let $\mathcal{L}_{Bag} = \langle \Sigma_{Bag}, \mathcal{V} \rangle$ be a language over a signature $\Sigma_{Bag} = \langle \mathcal{F}_{Bag}, \Pi \rangle$ such that the binary function symbol $\{\!\!\{\,\cdot\,|\,\cdot\,\}\!\!\}$ and the constant symbol `nil` are in $\mathcal{F}_{Bag}$, and $\Pi = \{=, \in\}$. A hybrid theory of multisets—called *Bag*—can be simply obtained from the theory of lists shown above. The constructor $[\,\cdot\,|\,\cdot\,]$ used for lists is replaced by the binary function symbol $\{\!\!\{\,\cdot\,|\,\cdot\,\}\!\!\}$. The behavior of this new symbol is regulated by the following equational axiom

$$\boxed{(E_p^m) \qquad \forall xyz\, \{\!\!\{\, x, y \,|\, z \,\}\!\!\} = \{\!\!\{\, y, x \,|\, z \,\}\!\!\}}$$

which, intuitively, states that the order of elements in a multiset is immaterial (*permutativity property*). Axioms (K), (W), (F_2), and (F_3) of *List*—with $[\,\cdot\,|\,\cdot\,]$ replaced by $\{\!\!\{\,\cdot\,|\,\cdot\,\}\!\!\}$ and $\mathcal{F}_{List}$ replaced by $\mathcal{F}_{Bag}$—still hold. Conversely, axiom schema (F_1) does not hold for multisets, when f is instantiated to $\{\!\!\{\,\cdot\,|\,\cdot\,\}\!\!\}$.

The same is true for compact lists and sets. Thus, in the general case—that is, assuming that also the symbols for compact lists and sets are introduced—axiom schema (F_1) is replaced by:

$$\boxed{\begin{array}{lll} (F_1') & \forall x_1 \cdots x_n y_1 \cdots y_n & \left(\begin{array}{c} f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n) \\ \to x_1 = y_1 \wedge \cdots \wedge x_n = y_n \end{array} \right) \\ & \multicolumn{2}{l}{\textit{for any } f \in \mathcal{F}_{Bag} \cup \mathcal{F}_{CList} \cup \mathcal{F}_{Set},} \\ & \multicolumn{2}{l}{f \textit{ distinct from } \{\!\!\{ \cdot \mid \cdot \}\!\!\},\ [\![\cdot \mid \cdot]\!],\ \{ \cdot \mid \cdot \}} \end{array}}$$

In $KWE_p^m F_1' F_2 F_3$, however, we lack in a general criterion for establishing equalities and disequalities between multisets. To obtain it, the following *multiset extensionality* property is introduced:

> *Two (hybrid) multisets are equal if and only if they have the same number of occurrences of each element, regardless of their order.*

The axiom proposed in [12] to force this property is the following:

$$\boxed{\begin{array}{lll} (E_k^m) & \forall y_1 y_2 v_1 v_2 & \left(\begin{array}{l} \{\!\!\{ y_1 \mid v_1 \}\!\!\} = \{\!\!\{ y_2 \mid v_2 \}\!\!\} \ \leftrightarrow \\ \qquad (y_1 = y_2 \wedge v_1 = v_2) \vee \\ \qquad \exists z\, (v_1 = \{\!\!\{ y_2 \mid z \}\!\!\} \wedge v_2 = \{\!\!\{ y_1 \mid z \}\!\!\}) \end{array} \right) \end{array}}$$

Observe that (E_k^m) implies (E_p^m). (E_k^m) is needed for establishing disequalities between bags.

2.3 Compact lists

Let $\mathcal{L}_{CList} = \langle \Sigma_{CList}, \mathcal{V} \rangle$ be a first-order language over a signature $\Sigma_{CList} = \langle \mathcal{F}_{CList}, \Pi \rangle$ such that the binary function symbol $[\![\cdot \mid \cdot]\!]$ and the constant symbol `nil` are in $\mathcal{F}_{CList}$, and $\Pi = \{=, \in\}$. Similarly to bags, a hybrid theory of *compact lists*—called *CList*—can be obtained from the theory of lists with only a few changes. The list constructor symbol is replaced by the binary function symbol $[\![\cdot \mid \cdot]\!]$, to be used as the compact list constructor. The behavior of this symbol is regulated by the equational axiom

$$\boxed{(E_a^c) \qquad \forall xy\, [\![x, x \mid y]\!] = [\![x \mid y]\!]}$$

which, intuitively, states that contiguous duplicates in a compact list are immaterial (*absorption property*). An example showing usefulness of compact lists comes from formal languages: let $s_1, \ldots, s_m, t_1, \ldots, t_n$ be elements of an alphabet, then

$$\begin{array}{ll} s_1^+ \cdots s_m^+ \text{ and } t_1^+ \cdots t_n^+ & \text{are the same regular expression} \\ \text{if and only if} & [\![s_1, \ldots, s_m]\!] = [\![t_1, \ldots, t_n]\!] . \end{array}$$

As for multisets, a general criterion for establishing both equality and disequality between compact lists is needed. This is obtained by introducing the following axiom:

$$\boxed{\begin{array}{lll} (E_k^c) & \forall y_1 y_2 v_1 v_2 & \left(\begin{array}{l} [\![y_1 \mid v_1]\!] = [\![y_2 \mid v_2]\!] \ \leftrightarrow \\ \qquad (y_1 = y_2 \wedge v_1 = v_2) \vee \\ \qquad (y_1 = y_2 \wedge v_1 = [\![y_2 \mid v_2]\!]) \vee \\ \qquad (y_1 = y_2 \wedge [\![y_1 \mid v_1]\!] = v_2) \end{array} \right) \end{array}}$$

Note that axiom (E_a^c) is implied by (E_k^c). Axioms (K), (W), (F_2)—with $[\cdot \mid \cdot]$ replaced by $[\![\cdot \mid \cdot]\!]$ and $\mathcal{F}_{List}$ replaced by $\mathcal{F}_{CList}$—and axiom (F_1') introduced for multisets, still hold. The freeness axiom (F_3), instead, needs to be suitably modified. As opposed to lists and multisets, an equation such as $X = [\![\, \mathtt{nil} \mid X \,]\!]$ admits a finite tree solution, namely a solution that binds X to the term $[\![\, \mathtt{nil} \mid t \,]\!]$, where t is any term. Therefore, axiom (F_3) is replaced by

$$\boxed{\begin{array}{ll} (F_3^c) & \forall x \quad (x \neq t[x]) \\ & \textit{unless } t \textit{ has the form } [\![\, t_1, \dots, t_n \mid x \,]\!], \\ & x \textit{ not occurring in } t_1, \dots, t_n, \textit{ and } t_1 = \cdots = t_n \end{array}}$$

2.4 Sets

Let $\mathcal{L}_{Set} = \langle \Sigma_{Set}, \mathcal{V} \rangle$ be a first-order language over a signature $\Sigma_{Set} = \langle \mathcal{F}_{Set}, \Pi \rangle$ such that the binary function symbol $\{\cdot \mid \cdot\}$ and the constant symbol `nil` are in $\mathcal{F}_{Set}$, and $\Pi = \{=, \in\}$. The last theory we consider is the simple theory of sets *Set*. Sets have both the *permutativity* and the *absorption properties* which, in the case of $\{\cdot \mid \cdot\}$, can be rewritten as follows:

$$\boxed{\begin{array}{llcl} (E_p^s) & \forall xyz\, \{x, y \mid z\} & = & \{y, x \mid z\} \\ (E_a^s) & \forall xy\, \{x, x \mid y\} & = & \{x \mid y\} \end{array}}$$

A criterion for testing equality (and disequality) between sets is obtained by merging the multiset equality axiom (E_k^m) and the compact list equality axiom (E_k^c):

$$\boxed{(E_k^s) \qquad \forall y_1 y_2 v_1 v_2 \quad \left(\begin{array}{r} \{y_1 \mid v_1\} = \{y_2 \mid v_2\} \;\leftrightarrow \quad\quad\quad\quad\quad\quad \\ (y_1 = y_2 \wedge v_1 = v_2) \vee \\ (y_1 = y_2 \wedge v_1 = \{y_2 \mid v_2\}) \vee \\ (y_1 = y_2 \wedge \{y_1 \mid v_1\} = v_2) \vee \\ \exists k\, (v_1 = \{y_2 \mid k\} \wedge v_2 = \{y_1 \mid k\}) \end{array} \right)}$$

According to (E_k^s) duplicates and ordering of elements in sets are immaterial. Thus, (E_k^s) implies the equational axioms (E_p^s) and (E_a^s). In [12] it is also proved that they are equivalent when domains are made by terms.

Axioms (K), (W), (F_2)—with $[\cdot \mid \cdot]$ replaced by $\{\cdot \mid \cdot\}$, and $\mathcal{F}_{List}$ replaced by $\mathcal{F}_{Set}$—and axiom (F_1') introduced for multisets, still hold. The modification of axiom (F_3) for sets, instead, simplifies the one used for compact lists:

$$\boxed{\begin{array}{ll} (F_3^s) & \forall x \quad (x \neq t[x]) \\ & \textit{unless } t \textit{ has the form } \{t_1, \dots, t_n \mid x\},\ x \textit{ not occurring in } t_1, \dots, t_n \end{array}}$$

Figure 1 summarizes the four theories. The two right-most axioms, Perm. (Permutativity) and Abs. (Absorption) are implied by (E_k^*) axioms and so they are actually superfluous. However, they are sufficient to characterize the theories from an equational point of view (see Section 3.1).

3 Privileged structures and solved form

In this section we briefly recall the privileged structures proposed in [12] for the four theories of the previous section. Then, we show that these structures and the theories correspond on the class of constraints analyzed. Moreover, we give a general notion of solved form that holds for constraints in all the four theories, and we show that a constraint in solved form is satisfiable in the corresponding privileged model (hence, in all the models of the theory, thanks to the correspondence result).

Name	empty	with	Equality		Herbrand	Acycl.	Perm.	Abs.
List	(K)	(W)	(F_1)		(F_2)	(F_3)		
Bag	(K)	(W)	(E_k^m)	(F_1')	(F_2)	(F_3)	●	
CList	(K)	(W)	(E_k^c)	(F_1')	(F_2)	(F_3^c)		●
Set	(K)	(W)	(E_k^s)	(F_1')	(F_2)	(F_3^s)	●	●

Figure 1: Axioms for the four theories

3.1 Privileged structures

In Section 2 we have presented four first-order hybrid theories for aggregates. For each of them, the behavior of a particular function symbol—the relevant aggregate constructor—is precisely characterized by an *equational theory*:

- E_{List}, the empty theory for *List*,
- E_{Bag}, the theory consisting of the *Permutativity* axiom (E_p^m) for *Bag*,
- E_{CList}, the theory consisting of the *Absorption* axiom (E_a^c) for *CList*,
- E_{Set}, the theory consisting of both the *Permutativity* (E_p^s) and *Absorption* (E_a^s) axioms for *Set*.

Using the appropriate equational theory we can define for each different kind of aggregate a privileged model for the relevant first-order theory. Let $\mathbb{T}$ be either *List* or *Bag* or *CList* or *Set*.

- The *domain* of the model is the quotient $T(\mathcal{F}_{\mathbb{T}})/\equiv_{\mathbb{T}}$ of the ordinary Herbrand Universe $T(\mathcal{F}_{\mathbb{T}})$ over the smallest congruence relation $\equiv_{\mathbb{T}}$ induced by the equational theory $E_{\mathbb{T}}$ on $T(\mathcal{F}_{\mathbb{T}})$.
- The interpretation of a term t is its equivalence class $[t]$.
- $=$ is interpreted as the identity on the domain $T(\mathcal{F}_{\mathbb{T}})/\equiv_{\mathbb{T}}$.
- The interpretation of membership is the following: $[t] \in [s]$ is `true` if and only if there is a term in $[s]$ of the form $[t_1, \dots, t_n, t \mid r]$ ($\{\!\{ t_1, \dots, t_n, t \mid r \}\!\}$, $[\![t_1, \dots, t_n, t \mid r]\!]$, or $\{t_1, \dots, t_n, t \mid r\}$) for some terms $t_1, \dots, t_n, r$.

Remark 3.1 *When $[s]$ is a multiset or a set, since the permutativity property holds, the requirement for $[t] \in [s]$ to be* `true` *can be simplified to: $[s]$ contains a term of the form $\{\!\{ t \mid r \}\!\}$ or $\{t \mid r\}$, respectively.*

These structures—named $\mathcal{LIST}, \mathcal{BAG}, \mathcal{CLIST}$, and $\mathcal{SET}$—are important models for the theories of aggregates we are studying, as it ensues from the following theorem.

Theorem 3.2 *The structures $\mathcal{LIST}, \mathcal{BAG}, \mathcal{CLIST}$, and $\mathcal{SET}$ and the theories List, Bag, CList, and Set correspond on the class of admissible constraints.*

3.2 Solved form

A particular form of constraints—called *solved form*—plays a fundamental rôle in establishing satisfiability of (general) constraints in the corresponding structures.

Definition 3.3 *A constraint C is in* pre-solved form *if all its literals are in pre-solved form, that is, they are in one of the following forms:*

- *$X = t$ and X does not occur neither in t nor elsewhere in C*
- *$t \in X$ and X does not occur in t*
- *$X \neq t$ and X does not occur in t*
- *$t \notin X$ and X does not occur in t.*

In order to establish satisfiability of a constraint in pre-solved form we need to introduce two further conditions that must be satisfied by the constraint, in particular by membership literals. If both conditions are satisfied we will say that the constraint is in *solved form.* Solved form constraints will be proved to be always satisfiable in the corresponding structure.

The first condition is informally motivated by the following example. Consider the constraint $X \in Y \wedge Y \in X$. It is in pre-solved form but it is clearly unsatisfiable in the structures $\mathcal{LIST}, \mathcal{BAG}, \mathcal{CLIST}$, and $\mathcal{SET}$. These constraints could be satisfiable in non well-founded models of membership. This topic is studied in [1] for equality constraints in the theory *Set*.

The first condition takes care of these situations and is precisely defined as follows. Let C be a pre-solved form constraint and $C^{\in}$ be the part of C containing only $\in$-atoms. Build the directed graph $\mathcal{G}_{C^{\in}}$ as follows:

nodes: Associate a distinct node to each $X \in FV(C^{\in})$.

edges: If $t \in X$ is in $C^{\in}$, $\nu_1, \dots, \nu_n$ are the nodes associated with the variables in t, and μ is the node associated with the variable X, then add the edges $\langle \nu_1, \mu \rangle, \dots, \langle \nu_n, \mu \rangle$.

A pre-solved form constraint C is *acyclic* if $\mathcal{G}_{C^{\in}}$ is acyclic.

The second condition for pre-solved form constraints is intuitively motivated by the following observations. Consider the constraint $a \in X \wedge a \notin X$. It is in pre-solved form and acyclic but unsatisfiable. Conversely, the constraint $\{A\} \in X \wedge \{a\} \notin X$ is satisfiable in $\mathcal{SET}$—take, for instance, any value of $A \neq a$ and $X = \{\{A\}\}$.

More in general, it is easy to see that whenever there are two literals $t \in X$ and $t' \notin X$ in C and t and t' unify in the considered theory $E_{\mathbb{T}}$ with the empty substitution ε, the constraint C is unsatisfiable. For example, the constraint $\{A, B\} \in X \wedge \{B, A\} \notin X$ in $\mathcal{L}_{Set}$ is unsatisfiable (indeed, terms $\{A, B\}$ and $\{B, A\}$ unify in *Set* with the empty substitution ε).

This condition, however, does not cover all the possible cases in which an acyclic constraint in pre-solved form is unsatisfiable, as it ensues from the following example. Let C be the $\mathcal{L}_{Set}$-constraint $a \in X \wedge X \in Y \wedge \{a \,|\, X\} \notin Y$. Observe that there are no pairs of terms t, t' of the form singled out above. Nevertheless, since $a \in X$ is equivalent to $\exists N\, (X = \{a \,|\, N\})$, by applying the substitution for X we get the pair of literals $\{a \,|\, N\} \in Y$ and $\{a, a \,|\, N\} \notin Y$. $\{a \,|\, N\}$ and $\{a, a \,|\, N\}$ unify in *Set* with ε: the latter constraint (hence, the former since it has been obtained by equivalent rewritings) is unsatisfiable.

To formally define the second condition for pre-solved constraints, taking into account all the possible cases informally described above, we introduce the following definitions and the subsequent lemma.

Definition 3.4 *Given a substitution $\theta \equiv [X_1/t_1, \ldots, X_n/t_n]$ and a natural number $m \geq 0$ we define by induction on m the substitution θ^m as:*

$$\begin{array}{rcl} \theta^0 & \equiv & \varepsilon \\ \theta^{m+1} & \equiv & [X_1/\theta^m(t_1), \ldots, X_n/\theta^m(t_n)] \end{array}$$

If there exists $m > 0$ such that $\theta^{m+1} \equiv \theta^m$ we say that θ is convergent*. Given a convergent substitution θ the* closure *θ^* of θ is the substitution θ^m such that $\forall k > m$ it holds $\theta^k \equiv \theta^m$.*

Definition 3.5 *Let C be a constraint in pre-solved form over the language $\mathcal{L}_{List}$ ($\mathcal{L}_{Bag}, \mathcal{L}_{CList}, \mathcal{L}_{Set}$) and let*

$$p_1^1 \in X_1, \ldots, p_1^{k_1} \in X_1, \ldots, p_q^1 \in X_q, \ldots, p_q^{k_q} \in X_q$$

be all the membership atoms of C. We define the member substitution *σ_C as follows:*

$$\sigma_C \equiv [X_1/[F_1, p_1^1, \ldots, p_1^{k_1} \,|\, M_1], \ldots, X_q/[F_q, p_q^1, \ldots, p_q^{k_q} \,|\, M_q]]$$

(resp., $\sigma_C \equiv [X_1/\{\!\!\{ F_1, p_1^1, \ldots, p_1^{k_1} \,|\, M_1 \}\!\!\}, \ldots,]$, $\sigma_C \equiv [X_1/[\![F_1, p_1^1, \ldots, p_1^{k_1} \,|\, M_1]\!], \ldots,]$, $\sigma_C \equiv [X_1/\{F_1, p_1^1, \ldots, p_1^{k_1} \,|\, M_1\}, \ldots,]$) where F_i and M_i are new variables not occurring in C.

Lemma 3.6 *If C is a constraint in pre-solved form and acyclic, and σ_C is its member substitution, then σ_C is convergent and $\sigma_C^* \equiv \sigma_C^{q-1}$, where q is the number of variables which occur in the right-hand side of membership atoms.*

As an example, let C be the pre-solved form and acyclic $\mathcal{L}_{Set}$-constraint

$$a \in Y \wedge Y \in X \wedge X \in Z \wedge \{\{a \,|\, Y\} \,|\, X\} \notin Z \tag{1}$$

It holds that:

$$\begin{array}{rcl} \sigma_C & \equiv & [Y/\{F_Y, a \,|\, M_Y\}, X/\{F_X, Y \,|\, M_X\}, Z/\{F_Z, X \,|\, M_Z\}], \\ \sigma_C^* & \equiv & [Y/\{F_Y, a \,|\, M_Y\}, X/\{F_X, \{F_Y, a \,|\, M_Y\} \,|\, M_X\}, \\ & & Z/\{F_Z, \{F_X, \{F_Y, a \,|\, M_Y\} \,|\, M_X\} \,|\, M_Z\}] \end{array}$$

We are now ready to introduce the definition of solved form.

Definition 3.7 *Let $E_{\mathbb{T}}$ be one of the four equational theories associated with the four kinds of aggregates. A constraint C in pre-solved form and acyclic is in* solved form *if for each pair of literals of the form $t \notin X, t' \in X$ in C we have that:*

$$E_{\mathbb{T}} \not\models \vec{\forall}(\sigma_C^*(t) = \sigma_C^*(t')).$$

The condition in the Definition 3.7 requires the ability to perform the test $E_{\mathbb{T}} \models \vec{\forall}(s = s')$ for any pair of terms s and s' in $\mathcal{L}_{\mathbb{T}}$. This test is connected with the availability of a unification algorithm for the theory $E_{\mathbb{T}}$. As a matter of fact, this test is equivalent to check if the empty substitution ε is a $E_{\mathbb{T}}$-unifier of $s = s'$. Since in [12] it is proved that the four theories we are dealing with are finitary (i.e., they admit a finite set of mgu's that covers all possible unifiers), this can be done using a unification algorithm for the theory at hand.

As an example, consider again the constraint (1). It holds that

$$\begin{array}{rcl}\sigma_C^*(X) & \equiv & \{F_X, \{F_Y, a \mid M_Y\} \mid M_X\} \\ \sigma_C^*(\{\{a \mid Y\} \mid X\}) & \equiv & \{\{a, F_Y, a \mid M_Y\}, F_X, \{F_Y, a \mid M_Y\} \mid M_X\}\end{array}$$

Hence, the constraint is not in solved form since $E_{Set} \models \vec{\forall}(\sigma_C^*(X) = \sigma_C^*(\{\{a \mid Y\} \mid X\}))$. Observe that using σ_C instead of σ_C^* the situation would not be detected, since $\{F_X, Y \mid M_X\} = \{\{a, F_Y, a \mid M_Y\}, F_X, Y \mid M_X\}$ is not satisfied, for instance, when $F_X \neq \{F_Y, a \mid M_Y\} \wedge Y \neq \{F_Y, a \mid M_Y\} \wedge \{F_Y, a \mid M_Y\} \notin M_X$.

Remark 3.8 *The solved form considered in [14], where only sets are taken into account, differs from the one considered in this paper in that the former does not include any atom of the form $t \in X$. As a matter of fact, in the theory Set it holds that (see also Remark* 3.1*) $s \in t \leftrightarrow \exists N(t = \{s \mid N\})$. Thus, in Set all membership constraints can be always replaced by equivalent equality constraints. This in turn implies that the additional conditions on the pre-solved form are not required at all. Cycle detection, for instance, is simply delegated to the unification algorithm used by the constraint rewriting procedure. The same holds also for multisets, but unfortunately it does not hold for lists and compact lists. In fact, $t \in X$ in the theory List (as well as in CList) cannot be replaced by a finite number of equality constraints. Therefore, since we want to have a single solved form which is adequate for all the four theories considered in this paper—in view of the combination of them into a single theory—we need to keep also atoms of the form $t \in X$ as irreducible constraints which can therefore occur in the solved form. Consequently, we added the further conditions on the literals to characterize solved forms in order to guarantee satisfiability.*

Theorem 3.9 *Let C_{List} (C_{Bag}, C_{CList}, C_{Set}) be a constraint in solved form over the language $\mathcal{L}_{List}$ (resp., $\mathcal{L}_{Bag}$, $\mathcal{L}_{CList}$, $\mathcal{L}_{Set}$). C_{List} (C_{Bag}, C_{CList}, C_{Set}) is satisfiable in $\mathcal{LIST}$ (resp., $\mathcal{BAG}$, $\mathcal{CLIST}$, $\mathcal{SET}$).*

4 Equality constraints

Equality constraints are conjunctions of atomic formulae based on the predicate symbol '=' (i.e., equations). Unification algorithms for verifying the satisfiability and producing the solutions of equality constraints in the four theories discussed in Section 2 have been proposed in [12]. They have been proved to terminate and to be sound and complete with respect to the corresponding axiomatic theories (namely, *List*, *Bag*, *CList*, and *Set*). It has been shown that the equality constraints are parametric with respect to these theories and that it is easy to merge them and to work in the combined theory that takes into account the four proposed data structures simultaneously.

The unification algorithms proposed in [12], namely:

- Unify_lists for lists,
- Unify_bags for multisets,

- Unify_clists for compact lists, and
- Unify_sets for sets,

will be used unaltered in the four global constraint solvers that we are going to propose in this paper (Section 6).

The output of the algorithms is either `false`, when the constraint is unsatisfiable, or a disjunction of solved form constraints (Def. 3.7) composed only by equality atoms. The complexity results for unification problems have been studied and proved to require linear time for lists, and to be NP-complete for the other forms of data aggregates.

5 Constraint rewriting procedures

We now extend the results presented in [12] for equality constraints to the whole classes of admissible constraints for the four constraint domains. We describe the constraint rewriting procedures used to eliminate all the literals not in pre-solved form possibly occurring in a given constraint C. We provide a different procedure for each kind of constraint literals—except for equality constraints whose constraint simplification procedures are constituted by the unification algorithms mentioned in the previous section. In the next section we will show how these procedures can be combined to test satisfiability, in the corresponding privileged structure, of any constraint written in one of the considered languages.

As done with the unification algorithms we will stress the parametric nature of all the procedures we define, by keeping their presentation as independent as possible from the kind of data aggregate they deal with. This will serve to let the merging of these procedures into a single general procedure be a straightforward step.

5.1 Lists

We begin the investigation with the theory *List*. If a constraint is a conjunction of equality atoms, then the decision problem for satisfiability can be solved in linear time, since it is simply a standard unification problem ([23, 24]).

If a constraint C is a conjunction of equalities and disequalities, then the satisfiability problem for *List* is still solvable in polynomial time $O(n^2)$ where $n = |C|$ [2, 9]. As far as disequalities are concerned, they can be managed by the procedure neq-list of Figure 2.

Lemma 5.1 *Let C be a constraint. Then List* $\models \vec{\forall}(C \leftrightarrow \text{neq-list}(C))$. *Moreover,* neq-list$(C)$ *can be implemented so as to run in time $O(n)$, where $n = |C|$.*

These polynomial results can not be extended to all the admissible constraints.

Theorem 5.2 *The satisfiability problem for conjunctions of $\in$-atoms and $\neq$-literals in List is NP-hard.*

We have seen how to reduce equality constraints (algorithm Unify_lists, Section 4) and disequality constraints (algorithm neq-list, Figure 2). In Figure 3 we show the

```
function neq-list($C$)
    while there is a $\neq$-constraint $c$ not in pre-solved form in $C$ do
    case $c$ of
(1)  $d \neq d$, $d$ is a constant                              $\mapsto$  false
(2)  $f(s_1,\dots,s_m) \neq g(t_1,\dots,t_n)$, $f \not\equiv g$   $\mapsto$  true
(3)  $t \neq X$, $t$ is not a variable                          $\mapsto$  $X \neq t$
(4)  $X \neq X$, $X$ is a variable                              $\mapsto$  false
(5)  $f(s_1,\dots,s_n) \neq f(t_1,\dots,t_n)$,
     $n > 0, f \not\equiv [\cdot\,|\,\cdot]$                    $\mapsto$  $s_1 \neq t_1 \vee$   $(i)$
                                                                           $\vdots$             $\vdots$
                                                                           $s_n \neq t_n$       $(n)$
(6)  $[s_1\,|\,s_2] \neq [t_1\,|\,t_2]$                          $\mapsto$  $s_1 \neq t_1 \vee$   $(i)$
                                                                           $s_2 \neq t_2$       $(ii)$
(7)  $X \neq f(t_1,\dots,t_n)$,
     $X \in FV(t_1,\dots,t_n)$                                  $\mapsto$  true
```

Figure 2: Rewriting procedure for disequations over lists

rewriting procedures in-list and nin-list for membership and negated membership literals over lists.

Theorem 5.3 *Given a constraint C, $\mathcal{LIST} \models \vec{\forall}(C \leftrightarrow$ nin-list(in-list(C))).*

In the proof of Theorem 5.3 no one of the axioms that distinguish the four theories is involved. Thus, the rewriting procedures for $\in$ and $\notin$ constraints over bags, compact lists and sets can be obtained from in-list and nin-list by replacing $[\cdot\,|\,\cdot]$ with the corresponding aggregate constructor symbol. When useful, we will refer to these procedures with the generic names in-$\mathbb{T}$ and nin-$\mathbb{T}$, where $\mathbb{T}$ is any of the considered theories.

Corollary 5.4 *Let C be a constraint and $\mathbb{T}$ be one of the theories Bag, CList, and Set. Then $\mathcal{A} \models \vec{\forall}(C \leftrightarrow$ nin-$\mathbb{T}$(in-$\mathbb{T}$(C))) where $\mathcal{A}$ is the structure corresponding to the theory $\mathbb{T}$.* $\square$

The following lemma will be useful to prove soundness and completeness of the global constraint solving procedure for *List*.

Lemma 5.5 *Let t, t' be two terms and C a solved form constraint over the language $\mathcal{L}_{List}$, such that $FV(t) \cup FV(t') \subseteq FV(C)$. If $\mathcal{LIST} \not\models \vec{\forall}(t = t')$, then $E_{List} \not\models \vec{\forall}(\sigma^*_C(t) = \sigma^*_C(t'))$.*

5.2 Multisets

We already know from [12] that the decision problem for multiset unification is NP-complete. Thus, the global satisfiability test is NP-hard. We know also that the

```
function in-list($C$)
    while there is a $\in$-constraint $c$ in $C$ not in pre-solved form do
        case $c$ of
(1)   $\left.\begin{array}{r} r \in f(t_1,\dots,t_n) \\ f \not\equiv [\cdot\,|\,\cdot] \end{array}\right\}$   $\mapsto$   false
(2)   $r \in [t\,|\,s]\ \}$   $\mapsto$   $r = t \vee$   $(a)$
                                          $r \in s$        $(b)$
(3)   $\left.\begin{array}{r} r \in X \\ X \in FV(r) \end{array}\right\}$   $\mapsto$   false
```

```
function nin-list($C$)
    while there is a $\notin$-constraint $c$ in $C$ not in pre-solved form do
        case $c$ of
(1)   $\left.\begin{array}{r} r \notin f(t_1,\dots,t_n) \\ f \not\equiv [\cdot\,|\,\cdot] \end{array}\right\}$   $\mapsto$   true
(2)   $r \notin [t\,|\,s]\ \}$   $\mapsto$   $r \neq t \wedge r \notin s$
(3)   $\left.\begin{array}{r} r \notin X \\ X \in FV(r) \end{array}\right\}$   $\mapsto$   true
```

Figure 3: Rewriting procedures for $\in$ and $\notin$ constraints over lists

same complexity results hold for compact list and set unification. Thus, the global satisfiability test will be NP-hard for all the considered data structures.

Equality constraints are managed by Unify_bags (see Section 4). Furthermore, thanks to Corollary 5.4, we know that the rewriting procedures in-list and nin-list developed for lists (see Figure 3) can be used almost unaltered also for bags.

As far as disequality constraints are concerned, a rewriting procedure—called neq-bag—capable of eliminating disequality constraints not in pre-solved form from the input constraint is presented in Figure 4.

In this procedure we make use of the functions tail and untail which are defined as follows:[1]

$$\begin{array}{lcll} \mathsf{tail}(f(t_1,\dots,t_n)) & = & f(t_1,\dots,t_n) & f \not\equiv \{\!\!\{\cdot\,|\,\cdot\}\!\!\} \\ \mathsf{tail}(X) & = & X & \\ \mathsf{tail}(\{\!\!\{t\,|\,s\}\!\!\}) & = & \mathsf{tail}(s) & \end{array} \qquad \begin{array}{lcl} \mathsf{untail}(X) & = & \mathtt{nil} \\ \mathsf{untail}(\{\!\!\{t\,|\,s\}\!\!\}) & = & \{\!\!\{t\,|\,\mathsf{untail}(s)\}\!\!\}\,. \end{array}$$

Special attention must be devoted to the management of disequalities between bags (rule (6.2) of neq-bag). If we use directly axiom (E_k^m), we have that:

$$\begin{array}{rcl} \{\!\!\{t_1\,|\,s_1\}\!\!\} \neq \{\!\!\{t_2\,|\,s_2\}\!\!\} & \leftrightarrow & (t_1 \neq t_2 \vee s_1 \neq s_2) \wedge \\ & & \forall N\,(s_2 \neq \{\!\!\{t_2\,|\,N\}\!\!\} \vee s_1 \neq \{\!\!\{t_1\,|\,N\}\!\!\}) \end{array}$$

An universal quantification is introduced: this is no longer a constraint according to our definition and, in any case, this is a quite complex formula to deal with.

Alternatively, we could use the intuitive notion of multi-membership: $x \in^i y$ if x belongs at least i times to the multiset y. This way, one can write an alternative

[1]Function tail is easily adapted to work with sets as well, assuming $\{\!\!\{\cdot\,|\,\cdot\}\!\!\}$ is replaced by $\{\cdot\,|\,\cdot\}$.

```
function neq-bag($C$)
    while there is a $\neq$-constraint $c$ in $C$ not in pre-solved form do
        case $c$ of
```

(1)—(5) as in neq-list			
(6.1)	$\{\!\!\{ t_1 \mid s_1 \}\!\!\} \neq \{\!\!\{ t_2 \mid s_2 \}\!\!\}$ tail(s_1) and tail(s_2) are the same variable	$\mapsto$	untail($\{\!\!\{ t_1 \mid s_1 \}\!\!\}$) $\neq$ untail($\{\!\!\{ t_2 \mid s_2 \}\!\!\}$)
(6.2)	$\{\!\!\{ t_1 \mid s_1 \}\!\!\} \neq \{\!\!\{ t_2 \mid s_2 \}\!\!\}$ tail(s_1) and tail(s_2) are not the same variable	$\mapsto$	$(t_1 \neq t_2 \wedge t_1 \notin s_2) \vee$ (a) $(\{\!\!\{ t_2 \mid s_2 \}\!\!\} = \{\!\!\{ t_1 \mid N \}\!\!\} \wedge s_1 \neq N)$ (b)
(7)	$X \neq t$ $X \not\equiv t, X \in FV(t)$	$\mapsto$	`true`

Figure 4: Rewriting procedure for $\neq$-constraints over bags

version of multiset equality and disequality. In particular, we have:

$$\begin{array}{rcl}\{\!\!\{ t_1 \mid s_1 \}\!\!\} \neq \{\!\!\{ t_2 \mid s_2 \}\!\!\} & \leftrightarrow & \exists X \exists n\, (n \in \mathbb{N} \wedge \\ & & (X \in^n \{\!\!\{ t_1 \mid s_1 \}\!\!\} \wedge X \notin^n \{\!\!\{ t_2 \mid s_2 \}\!\!\}) \vee \\ & & (X \in^n \{\!\!\{ t_2 \mid s_2 \}\!\!\} \wedge X \notin^n \{\!\!\{ t_1 \mid s_1 \}\!\!\} \{\!\!\{ t_2 \mid s_2 \}\!\!\}))\end{array}$$

In this case, however, we have a quantification on natural numbers: we are outside the language we are studying.

The rewriting rule shown in Figure 4 (rule (6.2)) avoids these difficulties introducing only existential quantification. Its correctness and completeness are proved in the following lemma.

Lemma 5.6 *Let C be a constraint. Then $\mathcal{BAG} \models \vec{\forall}(C \leftrightarrow \exists \bar{N}$ neq-bag$(C))$ where $\bar{N} = FV($neq-list$(C)) \setminus FV(C)$.*

Remark 5.7 *The procedure* in-bag *could safely be extended by the rule:*

(4)	$r \in X \quad \mapsto \quad X = \{\!\!\{ r \mid N \}\!\!\}$

where N is a new variable. One can add this rewriting rule, justified by the model $\mathcal{BAG}$, to reach a solved form that removes all occurrences of $\in$-constraints (see Remark 3.8) without affecting termination and completeness. As a matter of fact, none of the rewriting procedures Unify_bags, neq-bag, nin-bag *introduces $\in$-constraints. Thus, if we add rule (4) we are sure to completely remove $\in$-constraints from the constraints. Termination of this modified version of the algorithm follows trivially.*

The same considerations and results hold for sets but, as already observed in Remark 3.8, they do not hold for lists and compact lists. Therefore, when dealing with a theory at a time one could add rule (4) where appropriate. But when dealing with the global combined theory (see Section 7), since we assume that $\in$ is a polymorphic operator (i.e., it applies indistinctly to all the four types of data structures) and that there are no type declarations, we are no longer able to distinguish whether $t \in X$ can be rewritten using rule (4)—that is X is a set or a bag—or not. This is the

```
function neq-clist($C$)
    while there is a $\neq$-constraint $c$ in $C$ not in pre-solved form do
        case $c$ of
(1)—(5) as in neq-list
(6)    $\llbracket t_1 \mid s_1 \rrbracket \neq \llbracket t_2 \mid s_2 \rrbracket$ }  $\mapsto$
           $t_1 \neq t_2 \vee$                                                                        $(a)$
           $s_1 \neq s_2 \wedge \llbracket t_1 \mid s_1 \rrbracket \neq s_2 \wedge s_1 \neq \llbracket t_2 \mid s_2 \rrbracket$      $(b)$
(7.1)  $X \neq t$
       $X \in FV(t)$,
       $X$ is not $\llbracket t_1, \dots, t_n \mid X \rrbracket$,
       $n > 0$ and $X \notin FV(t_1, \dots, t_n)$   }  $\mapsto$  true
(7.2)  $X \neq \llbracket t_1, \dots, t_n \mid X \rrbracket$
       $X \notin FV(t_1, \dots, t_n)$   }  $\mapsto$  $t_1 \neq t_2 \vee$                       $(a.1)$
                                                    $\vdots$                                  $\vdots$
                                                    $t_1 \neq t_n \vee$                       $(a.n)$
                                                    $X = \texttt{nil} \vee$                   $(b)$
                                                    $X = \llbracket N_1 \mid N_2 \rrbracket \wedge N_1 \neq t_1$    $(c)$
```

Figure 5: Rewriting procedure for disequations over compact lists

reason why we prefer to not introduce this rule neither when dealing with bags and sets alone.

The following lemma will be useful to prove soundness and completeness of the global constraint solving procedure for *Bag*.

Lemma 5.8 *Let t, t' be two terms and C a solved form constraint over the language $\mathcal{L}_{Bag}$, such that $FV(t) \cup FV(t') \subseteq FV(C)$. If $\mathcal{BAG} \not\models \vec{\forall}(t = t')$, then $E_{Bag} \not\models \vec{\forall}(\sigma^*_C(t) = \sigma^*_C(t'))$.*

5.3 Compact Lists

As far as equality constraints are concerned, we can use the unification algorithm Unify_clists for compact lists (cf. Section 4). $\in$ and $\notin$ constraints are dealt with by the procedures in-clist and nin-clist trivially adapted from the same procedures for lists shown in Figure 3. It remains to deal with $\neq$-constraints. The rewriting procedure for this kind of constraints—called neq-clist—is shown in Figure 5.

Lemma 5.9 *Let $C_1, \dots, C_k$ be the constraints non-deterministically returned by* neq-clist(C) *and $\bar{N}_i = FV(C_i) \setminus FV(C)$. Then $\mathcal{CLIST} \models \vec{\forall}\left(C \leftrightarrow \bigvee_{i=1}^{k} \exists \bar{N}_i C_i\right)$.*

Observe that, differently from multisets, the rewriting rule for disequality of compact lists mimics perfectly the axiom (E^c_k). This has been possible since this axiom does not introduce (new) existentially quantified variables.

The following lemma will be useful to prove soundness and completeness of the global constraint solving procedure for *CList*.

```
function neq-set($C$)
    while there is a $\neq$-constraint $c$ in $C$ not in pre-solved form do
        case $c$ of
```

(1)—(5) as in neq-list				
(6)	$\{t_1 \mid s_1\} \neq \{t_2 \mid s_2\}$	$\mapsto$	$Z \in \{t_1 \mid s_1\} \wedge Z \notin \{t_2 \mid s_2\} \vee$	(a)
			$Z \in \{t_2 \mid s_2\} \wedge Z \notin \{t_1 \mid s_1\}$	(b)
(7.1)	$X \neq t$ $X \in FV(t)$, X is not $\{t_1, \ldots, t_n \mid X\}$, $n > 0$ and $X \notin FV(t_1, \ldots, t_n)$	$\mapsto$	`true`	
(7.2)	$X \neq \{t_1, \ldots, t_n \mid X\}$ $X \notin FV(t_1, \ldots, t_n)$	$\mapsto$	$t_1 \notin X \vee$	(i)
			$\vdots$	$\vdots$
			$t_n \notin X$	(n)

Figure 6: Rewriting procedure for disequations over sets

Lemma 5.10 *Let t, t' be two terms and C a solved form constraint over the language $\mathcal{L}_{CList}$, such that $FV(t) \cup FV(t') \subseteq FV(C)$. If $\mathcal{CLIST} \not\models \vec{\forall}(t = t')$, then $E_{CList} \not\models \vec{\forall}(\sigma^*_C(t) = \sigma^*_C(t'))$.*

5.4 Sets

The handling of equalities involving sets is governed by the unification algorithm Unify_sets (cf. Section 4). Procedures in-set and nin-set—adapted from the corresponding procedures for lists shown in Figure 3—are used for membership literals involving sets. The remaining constraints, namely, $\neq$-constraints, are managed by the rewriting procedure neq-set shown in Figure 6.

Some remarks are needed regarding rule (6). As for multisets, axiom (E^s_k) introduces an existentially quantified variable to state equality. Thus, its direct application for stating disequality requires universally quantified constraints that go outside the language.

The rewriting rule (6.2) used for multisets can not be used in this context. In fact, the property that $s_1 \neq N$ implies $\{\!\{ t_1 \mid s_1 \}\!\} \neq \{\!\{ t_1 \mid N \}\!\}$, that holds for finite multisets does not hold for sets. For instance, $\{a\} \neq \{a, b\}$ but $\{b, a\} = \{b, a, b\}$. Thus, this rewriting rule would be not correct for sets.

A rewriting rule for set-disequalities can be obtained by taking the negation of the standard extensionality axiom extended to deal with hybrid colored sets:

$$\boxed{\begin{array}{llll} (E_k) & x = y & \leftrightarrow & \forall z\,(z \in x \leftrightarrow z \in y) \wedge \\ & & & \ker(x) = \ker(y) \end{array}}$$

$\ker(t)$ identifies the kernel of a ground term t (operationally, it is the same as function tail of Section 5.2). Intuitively, $\ker(t)$ is what remains of a set when all its elements have been removed. In "standard" sets, $\ker(s) =$ `nil`. In colored sets, $\ker(s)$ can be any ground term of the form $f(t_1, \ldots, t_n)$, with $f \not\equiv \{\cdot \mid \cdot\}$ (axiom (K) ensures that such terms—called kernels—do not contain any element). For instance,

```
SAT_𝕋(C) = repeat
               C' := C;
               C := Unify_𝕋s(neq-𝕋(nin-𝕋(in-𝕋(C))));
           until C = C';
           return(is_solved_𝕋(C)).
```

Figure 7: The satisfiability procedure, parametric with respect to $\mathbb{T}$

$\ker(\{a \mid f(b)\}) = f(b)$. Axiom (E_k) has been proved in [12] to be equivalent to (E_k^s) in models whose domains are terms: in particular it holds in $\mathcal{SET}$.

This is the approach followed in [14]. Unfortunately this solution introduces some technical complications that require further special controls to check that a constraint possibly involving ker terms is satisfiable. We prefer to skip this issue here and refer the interested reader to [14].

In rule (6) of neq-set, therefore, we assume that all sets have the same kernel. If this would not be the case, then the neq-set procedure could be not correct. For example, $\{a \mid b\} \neq \{a \mid c\}$ is false according to rule (6), whereas it is true if also the kernels are taken into account. This simplification, however, is further motivated by the fact that in the combined theory (see Section 7) it will turn out to be convenient adding sorts to our underlying logic in order to avoid "mixed" aggregates—i.e., aggregates built using different aggregate constructors in the same term. The addition of sorts would provide also an immediate solution to the problem of colored sets, since sorts could force all sets to be based only on the empty set.

Lemma 5.11 *Let $C_1, \dots, C_k$ be the constraints non-deterministically returned by* neq-set(C) *and $\bar{N}_i = FV(C_i) \setminus FV(C)$. Then $\mathcal{SET} \models \vec{\forall}\left(C \leftrightarrow \bigvee_{i=1}^{k} \exists \bar{N}_i C_i\right)$, provided rule* (6) *is never fired by two terms with different* ker.

6 Constraint solving

In this section we address the problem of establishing if a constraint C written in one of the languages studied in this paper is satisfiable in the related privileged structure—and, thus, in any structure that models the corresponding theory.

We show how to produce solution constraints, namely, returning an equisatisfiable disjunction of solved form constraints—for each of the four theories.

Constraint satisfiability for the theory $\mathbb{T}$ is checked by the non-deterministic rewriting procedure $\mathsf{SAT}_{\mathbb{T}}$ shown in Figure 7. Its definition is completely parametric with respect to the theory involved. $\mathsf{SAT}_{\mathbb{T}}$ uses iteratively the various rewriting procedures presented in the previous section. Each disjunction generated by the rewriting rules of these procedures is interpreted as a (don't know) non-deterministic choice. Thus, $\mathsf{SAT}_{\mathbb{T}}(C)$ returns a collection $C_1, \dots, C_k$ of constraints. Each of them is either in solved form or `false`. The two conditions that guarantee that a constraint in pre-solved form is in solved form are tested by function $\mathsf{is\_solved}_{\mathbb{T}}$ shown in Figure 8. By Theorem 3.9 a constraint in solved form is guaranteed to be satisfiable in the corresponding structure.

```
function is_solved$_{\mathbb{T}}(C)$
    build the directed graph $\mathcal{G}_{C^{\in}}$
    if $\mathcal{G}_{C^{\in}}$ has a cycle
        then return false
        else
            compute $\sigma^*_C$
            if there is a pair $t \in X, t' \notin X$ in $C$ s.t. $\mathbb{T} \models \vec{\forall}(\sigma^*_C(t) = \sigma^*_C(t'))$
                then return false
                else return $C$.
```

Figure 8: Final check for solved form constraints

Theorem 6.1 (Termination) *Let $\mathbb{T}$ be one of the theories List, Bag, CList, and Set. Each non-deterministic execution of* $\mathsf{SAT}_{\mathbb{T}}(C)$ *terminates in a finite number of steps. Moreover, the constraint returned is either* `false` *or a solved form constraint.*

Lemma 6.2 *Let $\mathbb{T}$ be one of the theories List, CList, Bag and Set, and C a constraint in pre-solved form over the language of $\mathbb{T}$. If* is_solved$_{\mathbb{T}}$(C) *returns* `false`, *then C is not satisfiable in the structure $\mathcal{A}$ which corresponds to $\mathbb{T}$.*

Theorem 6.3 (Soundness and Completeness) *Let $\mathbb{T}$ be one of the theories List, Bag, CList, and Set, and $C_1, \ldots, C_k$ be the solved form constraints non-deterministically returned by* $\mathsf{SAT}_{\mathbb{T}}(C)$, *and $\bar{N}_i = FV(C_i) \setminus FV(C)$. Then $\mathcal{A} \models \vec{\forall}\left(C \leftrightarrow \bigvee_{i=1}^{k} \exists \bar{N}_i C_i\right)$, where $\mathcal{A}$ is the structure corresponding to the considered theory $\mathbb{T}$.*[2]

Corollary 6.4 *Given a constraint C, it is decidable whether $\mathcal{A} \models \vec{\exists} C$, where $\mathcal{A}$ is one of the structures $\mathcal{LIST}$, $\mathcal{BAG}$, $\mathcal{CLIST}$, $\mathcal{SET}$.*

7 Combining Theories

The four theories presented in the paper can be combined in order to provide more general frameworks where to deal with several of the proposed data structures simultaneously. As a matter of fact, the axioms of the four theories have been defined so as to make this combination a straightforward task. All the data structures are built in the same way (as regulated by axioms (W)), using the same kind of elements. Each axiom involves at most one aggregate constructor symbol, so that the theory for one aggregate is not influenced by the presence of axioms for the other aggregates. The combined theory is therefore obtained by simply taking the union of the sets of axioms of the four individual theories.

As regards the interpretation domain of the privileged structure for the combined theory a simple solution is obtained by: taking the union of the four equational theories considered in the individual cases (axioms (E_p^m) and (E_p^s), as well as axioms (E_a^c) and (E_a^s) must be considered different); taking the union of the set of terms for the individual theories; using the combined equational theory to compute the quotient of the combined Herbrand Universe $T(\mathcal{F}_{List} \cup \mathcal{F}_{Bag} \cup \mathcal{F}_{CList} \cup \mathcal{F}_{Set})$. This

[2]With the small exceptions for sets (see Lemma 5.11) that can however easily be overtaken.

simply causes some equivalence classes that are distinct in the individual cases to be merged in the same class in the combined case. Thus, for instance, terms $\{\!\{ a, b \}\!\}$ and terms $\{\!\{ b, a \}\!\}$ which are put into different classes if we consider only the equational axioms for *Set*, are instead members of the same equivalence class when considering the combined equational theory.

Theorem 3.9 ensures the satisfiability of a solved form constraint for all the theories: an effective way to find a successful valuation is given. It is easy to extend the result to the combined theory. The crucial point is that for variables X occurring only in constraints $X \neq t, t \notin X, t \in X$ the solution is found in $\mathcal{SET}$.

As regards constraint solving, also the various constraint rewriting procedures can be easily combined in order to obtain a general constraint solver for the combined theory. As a matter of fact, all rewriting rules used in these procedures have been obtained in a quite direct way from the relevant axioms and thus they inherit from the latter their parametric definition. Parametricity of the rewriting rules has been made evident throughout the presentation in previous sections. Specific instances of these rules are obtained by simply replacing one aggregate constructor with a different one. The global satisfiability procedure SAT for the combined case is obtained from the generic definition of $\mathsf{SAT}_{\mathbb{T}}$ (see Figure 7) by replacing each call to a generic procedure $p_{\mathbb{T}}(C)$ with the composition of the four specific calls $p_{Set}(p_{Bag}(p_{CList}(p_{List}(C))))$.

Since, for each theory $\mathbb{T}$, all the rewriting procedures do not generate any constraint not belonging to the theory itself, termination of the satisfiability procedure SAT for the combined case is immediately obtained from the termination of the satisfiability procedures for the individual theories. Similarly, soundness and completeness of the global satisfiability procedure is also preserved.

The language obtained by the combination of the four theories allows one to write terms that freely mix various kinds of different data structures. Thus, for instance, we can write a term like $\{a \mid [\![b, c]\!]\}$, which is in part constructed as a set and in part as a compact list.

To avoid the existence of such terms, for which it is hard to find a "natural" interpretation and which are likely to be of little practical utility, an elegant solution is to introduce a notion of sort—hence moving to the context of multi-sorted first-order languages.

Roughly speaking, in this context, one can associate a different sort with every symbol in the language. Thus, for instance, one can introduce the sort Set which is intuitively the sort of all the terms which denote sets. In the term $\{t_1 \mid t_2\}$, t_2 is required to be of sort Set, while t_1 can be of any sort. Thus, the sort of $\{\cdot \mid \cdot\}$ is $\mathsf{any} \times \mathsf{set} \to \mathsf{set}$. Only terms that respect their sorts are allowed to occur in admissible constraints.

This way, different data structures—i.e., data structures of different sorts—can not be mixed within the same term. Also the problem of colored aggregates disappears (provided a constant `nil` with the proper sort is assumed to exist for each distinct data structure, e.g., `nil`, $\{\!\{\ \}\!\}$, $[\![\ ]\!]$, $\emptyset$).

A detailed discussion of this topic is outside the scope of this paper. Indeed, the aim of this section is to show that the choices made in the axiomatic definition of the theories for the considered data aggregates, as well as the parametric definition of the relevant constraint rewriting procedures, make their combination into a single general framework immediately feasible, with only a very limited effort.

Conversely, turning this proposal into a concrete CLP programming language that provides all the four data structures altogether requires a few technical matters, such as those concerning the use of sorts, to be further refined.

8 Conclusions

In this paper we have extended the results of [12] studying the constraint solving problem for four different aggregate theories: the theories of lists, multisets, compact lists, and sets. The analyzed constraints are conjunctions of literals based on equality and membership predicate symbols. We have identified the privileged models for these theories by showing that they correspond with the theories on the set of admissible constraints. We have developed a notion of solved form (proved to be satisfiable) and presented the rewriting algorithms which allow to use this notion to decide the satisfiability problems in the four contexts.

In particular, we have shown how constraint solving can be developed parametrically for these theories and we have pointed out the differences and similarities between the four aggregate data structures. Moreover, we have faced complexity problems and we have discussed the issue of combining the independent results obtained.

As further work it could be interesting to analyze parametrically the behavior of the four data structures in presence of append-like operators (*append* for lists, $\cup$ for sets, $\uplus$ for multisets). It has been recently proved that these operators can not be defined without using universal quantifiers (or recursion) with the languages analyzed in this paper [11].

Acknowledgments

The authors wish to thank Ashish Tiwari and Silvia Monica for useful discussions on the topics of this paper. This work is partially supported by MURST project *Certificazione automatica di programmi mediante interpretazione astratta.*

References

[1] D. Aliffi, A. Dovier, and G. Rossi. From Set to Hyperset Unification. *Journal of Functional and Logic Programming*, 1999(10):1–48. The MIT Press, September 1999.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, 1998.

[3] J. Banatre and D. Le Metayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111. January 1993.

[4] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, vol. 96 (1992) 217-248.

[5] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set Constructors in a Logic Database Language. *Journal of Logic Programming 10*, 3 (1991), 181–232.

[6] D. Cantone, E. G. Omodeo, and A. Policriti. The Automation of Syllogistic. II. Optimization and Complexity Issues. *Journal of Automated Reasoning*, 6:173–187, 1990.

[7] C. C. Chang and H. J. Keisler. *Model Theory.* Studies in Logic. North Holland, 1973.

[8] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–321. Plenum Press, 1978.

[9] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In R.Mason ed., *Information Processing 1983*, Elevisier (North Holland), pp. 909–914.

[10] E. Dantsin and A. Voronkov. A Nondeterministic Polynomial-Time Unification Algorithm for Bags, Sets and Trees. In W. Thomas ed., *Foundations of Software Science and Computation Structure*, LNCS Vol. 1578, pages 180–196, 1999.

[11] A. Dovier, C. Piazza, and A. Policriti. Comparing expressiveness of set constructor symbols. In H. Kirchner and C. Ringeissen, eds., *FROCOS'00*, LNCS No. 1794, pp. 275–289, 2000.

[12] A. Dovier, A. Policriti, and G. Rossi. A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. *Fundamenta Informaticae*, 36(2/3):201–234, 1998.

[13] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1–44, 1996.

[14] A. Dovier and G. Rossi. Embedding Extensional Finite Sets in CLP. In D. Miller, editor, *Proc. of International Logic Programming Symposium, ILPS'93*, pages 540–556. The MIT Press, Cambridge, Mass., October 1993.

[15] H. B. Enderton. *A mathematical introduction to logic.* Academic Press, 1973. 2^{nd} printing.

[16] C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1:191–246, 1997.

[17] S. Grumbach and T. Milo. Towards tractable algebras for bags. *Journal of Computer and System Sciences*, 52(3):570–588, 1996.

[18] P. M. Hill, and J. W. Lloyd. *The Gödel Programming Language.* The MIT Press, Cambridge, Mass., 1994.

[19] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19–20:503–581, 1994.

[20] D. Kapur and P. Narendran. NP-Completeness of the Set Unification and Matching Problems, In J. H. Siekmann ed., 8th CADE, LNCS n. 230, pp. 489–495, 1986.

[21] K. Kunen. *Set Theory. An Introduction to Independence Proofs.* Studies in Logic. North Holland, 1980.

[22] A. Mal'cev. Axiomatizable Classes of Locally Free Algebras of Various Types. In *The Metamathematics of Algebraic Systems*, Collected Papers, Ch. 23. North Holland, 1971.

[23] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[24] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer System Science*, 16(2):158–167, 1978.

[25] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*, Second Edition. Prentice Hall, 1996.

[26] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets, an introduction to SETL.* Springer-Verlag, Berlin, 1986.

[27] J. H. Siekmann. Unification theory. In C. Kirchner, editor, *Unification.* Academic Press, 1990.

[28] A. Tzouvaras. The Linear Logic of Multisets. *Logic Journal of the IGPL*, Vol. 6, No. 6, pp. 901–916, 1998.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 100 - 111.

Membrane computing based on splicing: improvements

Pierluigi Frisco
LIACS, Leiden University,
Niels Bohwerg 1, 2333 CA Leiden, The Netherlands
e-mail: pier@liacs.nl

Abstract. New computability models, called super-cell systems or P systems, based on the evolution of objects in a membrane structure, were recently introduced. The seminal paper of Gheorghe Păun describes three ways to look at them: transition, rewriting and splicing super-cell systems having different properties.
Here we investigate two variants of splicing P systems improving results concerning their generative capability. This is obtained with a variant of the "rotate-and-simulate" technique classical in H systems area.

1 Introduction

P systems were recently introduced in [5] as distributed parallel computing models.

In the seminal paper the author considers systems based on a hierarchical arranged, finite *cell-structure* consisting of several cell-membranes embedded in a main membrane called *skin*. The membranes delimit *regions* where *objects*, elements of a finite set or alphabet, are placed.

The objects evolve according to given *evolution rules* associated with a region; priorities can be associated to evolution rules. They contain symbols as a_{here}, a_{out} or a_{in_i} where a is an object. The meaning of the subscripts is: *here* indicates that the object remains in the membranes in which it was produced; *out* means that that the object in sent out of the membranes in which it was produced; in_i means that the object is sent to membrane i if it is reachable from the region where the rule is applied, if not the rule is not applied.

The objects can evolve independently or in cooperation with the other objects present in the region in which it is. An evolution rule can destroy the membrane in which it is. In this case all the objects of the destroyed membrane pass to the immediately superior one and they evolve according to this one's evolution rules. The rules of the dissolved cell are lost. The skin membrane cannot be dissolved.

Such a system evolves in parallel: at each step all objects which can evolve should do it. A *computation* starts from an initial configuration of a system, defined by a cell-structure with objects and evolution rules in each cell, and terminates when no further rule can be applied.

It is possible to assign a result to a computation in two ways: considering the multiplicity of objects present in a designed membrane in a halting configuration, or concatenating the symbols leaving the system in the order they are sent out of the skin membrane.

In [5] the author examines three ways to look at P systems: transition, rewriting and splicing super-cell systems. Starting from these several variants were considered:

[6] gives a survey; in [7] polarized membranes and "electrical charges" assigned to objects are considered; in [10] rules with a_{in} (indicating that an object passes to any of the adjacent lower membranes non-deterministically chosen) and other types of structures (planar maps described by asymmetric graphs) are introduced; in [11] variants of splicing P systems with or without planar map are investigated. In most of the cases the characterization of recursively enumerable (RE) number relations or representation of permutation closures of RE languages are obtained.

We focused our attention on some of the systems introduced and studied in [11]. The objects of our investigations are P systems using string-object evolving by splicing with non-deterministic way of communicating and P system using string-object evolving by splicing working on planar maps described by asymmetric graphs. The characterization of RE languages is improved reducing the degree and the depth of the systems. One minimal result is obtained.

2 Splicing and P systems

The operation of splicing as a formal model of DNA recombination with the presence of restriction enzymes and ligases was introduced in [2]. Now we give definitions strictly related with our work; more general information may be found in [9].

Consider an alphabet V and two special symbols, # and \$ not in V. With V^* we indicate the free monoid generated by by the alphabet V under the operation of concatenation; λ indicates the empty string; the length of $x \in V^*$ is indicated with $|x|$.

A *splicing rule* is a string of the form $r = u_1\#u_2\$u_3\#u_4$, where $u_1, u_2, u_3, u_4 \in V^*$. For such a splicing rule r and strings $x, y, z, w \in V^*$ we write:

$$(x, y) \vdash_r (z, w) \quad \text{iff} \quad \begin{aligned} & x = x_1u_1u_2x_2, y = y_1u_3u_4y_2, \\ & z = x_1u_1u_4y_2, w = y_1u_3u_2x_2, \\ & \text{for some } x_1, x_2, y_1, y_2 \in V^*. \end{aligned} \tag{1}$$

What just defined is called *2-splicing* as two strings, z and w, are obtained as output. For a 2-splicing we call z and w the first and the second output string respectively.

In (1) it is also possible to consider only z as output. In this case the operation is called *1-splicing*.

Considering a rule r as the one defined above it is possible to create $r' = u_3\#u_4\$u_1\#u_2$ so that:

$$(y, x) \vdash_{r'} (w, z) \quad \text{iff} \quad \begin{aligned} & x = x_1u_1u_2x_2, y = y_1u_3u_4y_2, \\ & z = x_1u_1u_4y_2, w = y_1u_3u_2x_2, \\ & \text{for some } x_1, x_2, y_1, y_2 \in V^*. \end{aligned} \tag{2}$$

where $x, y, z, w, u_1, u_2, u_3, u_4 \in V^*$.

Based on 2-splicing the notion of an *H scheme* can be defined as a pair $\sigma = (V, R)$ where V is an alphabet and $R \subseteq V^*\#V^*\$V^*\#V^*$ is a set of splicing rules. For an

H scheme and a language $L \subseteq V^*$ we define

$$\begin{aligned}\sigma(L) &= \{z \in V^* \mid (x,y) \vdash_r (z,w) \text{ or } (x,y) \vdash_r (w,z),\\ &\qquad\qquad \text{for some } x,y \in L, r \in R, w \in V^*\},\\ \sigma^0(L) &= L,\\ \sigma^{i+1}(L) &= \sigma^i(L) \cup \sigma(\sigma^i(L)),\ i \geq 0,\\ \sigma^*(L) &= \bigcup_{i\geq 0} \sigma^i(L).\end{aligned}$$

The *diameter* of σ (the concept of diameter was introduced in [3] where it was called width) is indicated by $dia(\sigma) = (n_1, n_2, n_3, n_4)$, where

$$n_i = \max\{|u_i| \mid u_1\#u_2\$u_3\#u_4 \in R\},\ 1 \leq i \leq 4. \tag{3}$$

If we consider two families of languages FL_1 and FL_2, we define:

$$H(FL_1, FL_2) = \{\sigma^*(L) \mid L \in FL_1 \text{ and } \sigma = (V,R), R \in FL_2\}.$$

We denote by FIN, REG the families of finite and of regular languages respectively. We have (see details in [9])

$$FIN \subset H(FIN, FIN) \subset REG.$$

An *extended H system* is a construct $\gamma = (V, T, A, R)$, where V and T are alphabets so that $T \subseteq V$ (T is called *terminal* alphabet), A is a language on V (A is the set of *axioms*), and R is a set of splicing *rules* over V. The language generated by γ is $L(\gamma) = \sigma^*(A) \cap T^*$. The diameter of an extended H system γ (indicated by $dia(\gamma) = (n_1, n_2, n_3, n_4)$) is defined in a way similar to (3).

It is known by [1] and [12] that extended H systems with finite sets of axioms and splicing rules characterize REG.

A *splicing P system* of *degree* $m, m \geq 1$, is a construct

$$\Pi = (V, T, \mu, L_1, \cdots, L_m, R_1, \cdots, R_m),$$

where V is an alphabet; $T \subseteq V$ is the terminal alphabet; μ is a membrane structure consisting of m membranes labeled in a one-to-one manner with $1, \cdots, m$; $L_i \subseteq V^*, 1 \leq i \leq m$ are languages associated with the regions $1, \cdots, m$ of μ; $R_i, 1 \leq i \leq m$, are finite sets of evolution rules associated with the regions $1, \cdots, m$ of μ, of the following form: (r, tar_1, tar_2), where $r = u_1\#u_2\$u_3\#u_4$ is a 2-splicing rule over V, $\#, \$ \notin V$ and $tar_1, tar_2 \in \{here, out, in\}$ are called *target indication.*

A *configuration* of Π is an m-tuple $(M_1, \cdots, M_m)$ of languages over V. For two configurations $(M_1, \cdots, M_m), (M'_1, \cdots, M'_m)$ of Π we write $(M_1, \cdots, M_m) \Rightarrow (M'_1, \cdots, M'_m)$ if it is possible to pass from $(M_1, \cdots, M_m)$ to $(M'_1, \cdots, M'_m)$ applying in parallel the splicing rules of each membrane of μ to all possible strings of the corresponding membrane. So for $0 \leq i \leq m$ if $x = x_{i1}u_{i1}u_{i2}x_{i2}, y = y_{i1}u_{i3}u_{i4}y_{i2} \in M_i$ and $(r = u_{i1}\#u_{i2}\$u_{i3}\#u_{i4}, tar_{i1}, tar_{i2}) \in R_i, x_{i1}, x_{i2}, y_{i1}, y_{i2}, u_{i1}, u_{i2}, u_{i3}, u_{i4} \in V^*$, we have $(x,y) \vdash_r (z,w)$, $z, w \in V^*$. The strings z and w will go to the regions indicated by tar_{i1} and tar_{i2} respectively. For $j = 1, 2$, if $tar_{ij} = here$ then the string remains in membrane i; if $tar_{ij} = out$ the string is moved to the region immediately outside membrane i (if i if the skin membrane the string leaves the system); if $tar_{ij} = in$ the string is moved to any region immediately below membrane i. Note that as strings are supposed to appear in arbitrary many copies, after the application of rule r in a membrane i the strings x and y are still available in the same region, but if a string is sent out of a membrane then no copy of it remains here.

A *computation* is a sequence of transitions between configurations of a system Π starting from the initial configuration $(L_1, \cdots, L_m)$. The result of a computation is given by all strings in T^* the skin membrane sends out. All strings of this type define the language generated by Π and it is indicated by $L(\Pi)$.

Note that if a string is sent out of the system but it is not entirely made of symbols in T it is ignored, on the other hand a string in the system composed only by symbols in T does not contribute to the generated language.

The *depth* of a P system is defined by the height of the tree describing its membrane structure.

The *diameter* of a splicing P system $\Pi = (V, T, \mu, L_1, \cdots, L_m, R_1, \cdots, R_m)$, indicated by $dia(\Pi) = (n_1, n_2, n_3, n_4)$, is defined by

$$n_i = \max\{|u_i| \mid u_1\#u_2\$u_3\#u_4 \in R_1 \cup \cdots \cup R_m\},\ 1 \leq i \leq 4. \tag{4}$$

We denote by $SPL(i/o, m, p, (n_1, n_2, n_3, n_4))$ the family of languages $L(\Pi)$ generated by splicing P systems as above of degree at most $m, m \geq 1$, depth $p, p \geq 1$ and diameter (n_1, n_2, n_3, n_4).

It is possible to generalize the description of a P system passing from a tree structure to a graph (different from a tree) structure. An *asymmetric planar graph* is so made that for each two nodes i, j there is at most one of $(i, j), (j, i)$ edges. Such a graph is a representation of a planar map such that each border segment can be crossed in one direction only.

A *splicing P system on asymmetric graph* of degree $m, m \geq 1$, is a construct

$$\Pi = (V, T, g, L_1, \cdots, L_m, R_1, \cdots, R_m),$$

where $V, T, L_1, \cdots, L_m, R_1, \cdots, R_m$ are similar to the ones defined for a splicing P system of degree m. The only difference is that $tar_{ij} \in \{here, out, go\}, 1 \leq i \leq m, j = 1, 2$, where *here* and *out* have the same effect as described for splicing P systems, and *go* indicates that the string must go to another room non-deterministically chosen among the ones to which the string can move through a wall which permits communications. The set g defines couples indicating the edges of the graph having $L_1, \cdots, L_m$ as nodes. So g defines the permitted communication between the membranes in Π.

The *diameter* of a splicing P system on asymmetric graph Π (indicated by $dia(\Pi)$) is defined in a way similar to (4).

We denote by $SP'L(go, m, (n_1, n_2, n_3, n_4))$ the family of languages $L(\Pi)$ generated by splicing P systems on asymmetric graph as above of degree at most $m, m \geq 1$, and diameter (n_1, n_2, n_3, n_4).

In the next two sections we demonstrate theorems regarding the generative power of splicing P systems and splicing P systems on asymmetric graphs. These theorems represent an improvement of results present in [11] and [4].

3 Splicing P systems

In [11] the authors demonstrate that $SPL(i/o, 3, 3) = RE$ (Theorem 1) and that $SPL(i/o, 5, 2) = RE$ (Theorem 3). Both systems used for the proofs have (1, 2, 2, 1) as diameter. In [4] the authors show that $SPL(i/o, 2, 2, (2, 2, 2, 2)) = RE$ (Theorem 1). Hereby, using a variant of the "rotate-and-simulate" technique introduced in [8], we demonstrate that it is possible to have a splicing P system generating RE keeping the degree of the system equal to 2 (so as a consequence also the depth is 2) and the diameter equal to (1, 2, 2, 1).

Theorem 1 $SPL(i/o, 2, 2, (1, 2, 2, 1)) = RE$

Proof. Let $G = (N, T, S, R)$ be a type-0 Chomsky grammar in Kuroda normal form (this means that the productions in R can be of the forms $A \to a, A \to CD, AC \to DE$ or $A \to \lambda$ where $A, C, D, E \in N$ and $a \in T$) and B be a symbol not in $N \cup T$. Let us assume that symbols in $N \cup T \cup \{B\}$ can be numbered in a one-to-one manner so that $N \cup T \cup \{B\} = \{\alpha_1, \cdots, \alpha_n\}$ and that R contains m productions: $u_i \to v_i, 1 \leq i \leq m$. Moreover R can be divided in two sets: $R_1 = \{u_i \to v_i \mid u_i \to v_i \in R \wedge |u_i| = 1\}$ and $R_2 = \{u_i \to v_i \mid u_i \to v_i \in R \wedge |u_i| = 2\}$ so that $R_1 \cup R_2 = R$ and $R_1 \cap R_2 = \emptyset$. Consider also $R' = \{u \to u \mid u \in \{\alpha_1, \cdots, \alpha_n\}\}$ and that $\{o, X, X_1, X_2, Y, Y_1, Y_2, Z_{X_1}, Z_{X_2}, Z_Y, Z_{Y_2}, Z_\lambda, Z'_\lambda\} \cup \{Z_{X_i}, Z_{Y_i} \mid 1 \leq i \leq n + m\} \cup \{Y'_i, Z_{Y'_i} \mid u_i \to v_i \in R_2\}$ are symbols not in $N \cup T$.

Hereby the splicing P system of degree 2, depth 2 and diameter (1, 2, 2, 1) simulating the just defined grammar is described. For a better understanding of the demonstration splicing rules are numbered.

$\Pi = \{V, T, \mu, L_1, L_2, R_1, R_2\}$,
$V = N \cup T \cup \{o, B, X, X_1, X_2, Y, Y_1, Y_2, Z_{X_1}, Z_{X_2}, Z_Y, Z_{Y_2}, Z_\lambda, Z'_\lambda\} \cup$
$\{Z_{X_i}, Z_{Y_i} \mid 1 \leq i \leq n + m\} \cup \{Y'_i, Z_{Y'_i} \mid u_i \to v_i \in R_2\}$,
$\mu = [_1 [_2]_2]_1$,
$L_1 = \{XBSY, X_2 Z_{X_2}, Z_{Y_1} Y_1, X Z_X, Z_\lambda, Z'_\lambda\} \cup \{Z_{Y_i} o^i Y_1 \mid 1 \leq i \leq n + m\} \cup$
$\{Z'_{Y_i} Y'_i \mid u_i \to v_i \in R_2\}$,
$L_2 = \{Z_{Y_2} Y_2, X_1 Z_{X_1}, Z_Y Y\} \cup \{X_1 o^i v_i Z_{X_i} \mid 1 \leq i \leq n + m\}$,
$R_1 = \{1)(\#u_i Y\$Z_{Y_i}\#; in, out) \mid 1 \leq i \leq n + m\} \cup$
$\{2)(\#CY\$Z_{Y'_i}\#; here, out), 3)(\#AY'_i\$Z_{Y_i}\#; in, out) \mid u_i \to v_i \in R_2\} \cup$
$\{4)(\#Z_{X_2}\$X_1\#o; here, out), 5)(\#oY_2\$Z_{Y_1}\#; in, out), 6)(\#Z_X\$X_1\#\alpha; in, out)$,
$7)(\#BY\$Z_\lambda\#; here, out), 8)(\#Z'_\lambda\$X\#; out, out) \mid \alpha \in N \cup T \cup \{B\}\}$,
$R_2 = \{9)(\#Y_1\$Z_{Y_2}\#; here, out), 10)(\#Z_{X_i}\$X\#; out, out), 11)(\#Z_{X_1}\$X_2 o\#; out, out)$,
$12)(\alpha\#Y_2\$Z_Y\#; out, out) \mid 1 \leq i \leq n + m, \alpha \in N \cup T \cup \{B\}\}$.

During the subsequent demonstration note that all second output strings do not have any active role in the system, so Π could be based on 1-splicing.

The idea of the proof is based on the "rotate-and-simulate" technique, classic in H systems area. The sentential forms generated by G are simulated in Π in a circular permutation $Xw_1Bw_2Y, w_1, w_2 \in \{N \cup T\}^*$, with variants of X and Y. They will be present in a membrane of Π if and only if w_2w_1 is a sentential form of G. It is possible to remove the nonterminal symbol Y only with B from strings of the form $XwBY$. In this way the correct permutation of the string is ensured.

The simulation of a production in R and the rotation are done in the same way.

Assume that in membrane 1 we have a string of the form Xwu_iY with $w, u_i \in \{N \cup T \cup \{B\}\}^*$ (initially we have $XBSY$).

If a production in $R_1 \cup R'$ is simulated we have $(Xw \mid u_iY, Z_{Y_i} \mid o^iY_1) \vdash_1 (Xwo^iY_1, Z_{Y_i}u_iY)$ the first output string is sent into membrane 2 while the second is sent out of the system.

If a production in R_2 is simulated we have $(XwA \mid CY, Z_{Y_i'} \mid Y_i') \vdash_2 (XwAY_i', Z_{Y_i'}CY)$ (the first output string remains in membrane 1 and the second leaves the system) and then $(Xw \mid AY_i', Z_{Y_i} \mid o^iY_1) \vdash_3 (Xwo^iY_1, Z_{Y_i}AY_i'), 1 \leq i \leq n+m$ (the first output string is sent to membrane 2 and the second leaves the system).

In both cases the suffix u_iY is changed with $o^iY_1, 1 \leq i \leq n+m$. The strings leaving the system do not belong to T^* so they do not contribute to the language generated by Π.

In membrane 2, with a string as Xwo^iY_1, it is possible to perform $(Xwo^i \mid Y_1, Z_{Y_2} \mid Y_2) \vdash_9 (Xwo^iY_2, Z_{Y_2}Y_1)$. The second output string is sent to membrane 1 where no splicing rule can be applied; the string Xwo^iY_2, remaining in membrane 2, can be spliced so to have $(X_1o^jv_j \mid Z_{X_j}, X \mid wo^iY_2) \vdash_{10} (X_1o^jv_jwo^iY_2, XZ_{X_j}), 1 \leq j \leq n+m$. Both output strings are sent to membrane 1 but only the first one can be involved in splicing operations. A string as Xwo^iY_1 can also be spliced in membrane 2 by rule 10 so to have: $(X_1o^jv_j \mid Z_{X_j}, X \mid wo^iY_1) \vdash_{10} (X_1o^jv_jwo^iY_1, XZ_{X_j}), 1 \leq j \leq n+m$. Both output strings are sent to membrane 1. The second one cannot be involved in any splicing, with the first it is possible to have $(X_2 \mid Z_{X_2}, X_1 \mid o^jv_jwo^iY_1) \vdash_4 (X_2o^jv_jwo^iY_1, X_1Z_{X_2})$ but both strings, remaining in membrane 1, are no longer spliced.

A string of the form $X_1o^jv_jwo^iY_2$ can be spliced in membrane 1 so to substitute X_1 with X_2 and oY_2 with Y_1. This happens by $(X_2 \mid Z_{X_2}, X_1 \mid o^jv_jwo^iY_2) \vdash_4 (X_2o^jv_jwo^iY_2, X_1Z_{X_2})$ (the first output string remains in membrane 1 while the second is sent out of the system) and $(X_2o^jv_jwo^{i-1} \mid oY_2, Z_{Y_1} \mid Y_1) \vdash_5 (X_2o^jv_jwo^{i-1}Y_1, Z_{Y_1}oY_2)$ (the first output string is sent in membrane 2 while the second leaves the system). In membrane 1 it is also possible to have $(X_1o^jv_jwo^{i-1} \mid oY_2, Z_{Y_1} \mid Y_1) \vdash_5 (X_1o^jv_jwo^{i-1}Y_1, Z_{Y_1}oY_2)$. The second output string is sent out of the system while the first to membrane 2. Here this last string can be spliced so to have $(X_1o^jv_jwo^{i-1} \mid Y_1, Z_{Y_2} \mid Y_2) \vdash_9 (X_1o^jv_jwo^{i-1}Y_2, Z_{Y_2}Y_1)$. The first output string remains in membrane 2, the second is sent to membrane 1 and both cannot be involved in any splicing operation. The strings sent out of the system do not belong to T^* so they do not contribute to the language generated by Π.

In membrane 2 a string as $X_2 o^j v_j w o^{i-1} Y_1$ can be spliced so to substitute Y_1 with Y_2 and $X_2 o$ with X_1. This is obtained by $(X_2 o^j v_j w o^{i-1} \mid Y_1, Z_{Y_2} \mid Y_2) \vdash_9 (X_2 o^j v_j w o^{i-1} Y_2, Z_{Y_2} Y_1)$ (the first string remains in membrane 2, the second is sent to membrane 1 and cannot be involved in any splicing) and $(X_1 \mid Z_{X_1}, X_2 o \mid o^{j-1} v_j w o^{i-1} Y_2) \vdash_{11} (X_1 o^{j-1} v_j w o^{i-1} Y_2, X_2 o Z_{X_1})$ (both output strings are sent to membrane 1 but only the first one can be spliced). In membrane 2 it is also possible to have $(X_1 \mid Z_{X_1}, X_2 o \mid o^{j-1} v_j w o^{i-1} Y_1) \vdash_{11} (X_1 o^{j-1} v_j w o^{i-1} Y_1, X_2 o Z_{X_1})$. Both strings are sent to membrane 1 but only the first one can be spliced with $X_2 Z_{X_2}$ by rule 4 so to obtain $X_2 o^{j-1} v_j w o^{i-1} Y_1$, remaining in membrane 1 and no more spliced, and $X_1 Z_{X_2}$ not in T^* sent out of the system.

The process of decreasing the number of o's on the left and on the right of strings goes on between membranes 1 and 2. At a certain point three kinds of strings can be present: $X_1 v_j w Y_2, X_1 o^k v_j w Y_2$ in membrane 1 and $X_2 v_j w o^k Y_1$ in membrane 2, $1 \leq k \leq n+m-1$.

As described before a string as $X_1 o^k v_j w Y_2$ can be spliced with $X_2 Z_{X_2}$ by rule 4 so to obtain $X_2 o^{k-1} v_j w Y_2$, remaining in membrane 1 and no more spliced, and $X_1 Z_{X_2} \notin T^*$ sent out of the system.

In membrane 2 a string as $X_2 v_j w o^k Y_1$ can change the suffix oY_1 with Y_2 by rule 9 and the string $Z_{Y_2} Y_2$. The output strings $X_2 v_j w o^{k-1} Y_2$, remaining in membrane 2, and $Z_{Y_2} Y_1$, sent in membrane 1, are no longer used.

The string $X_1 v_j w Y_2$ can be spliced in membrane 1 so that $(X \mid Z_X, X_1 \mid v_j w Y_2) \vdash_7 (X v_j w Y_2, X_1 Z_X)$. The first output string is sent to membrane 2 while the second (not in T^*) out of the system. In membrane 2 it is possible to have $(X v_j w \mid Y_2, Z_Y \mid Y) \vdash_{12} (X v_j w Y, Z_Y Y_2)$. Both output strings are sent to membrane 1 but only the first one can get involved in splicing operations.

What it was just described is the process to pass from $X w u_i Y$ to $X v_j w Y$ simulating a production in R or rotating the substring between X and Y with one symbol.

At any moment a string of the form XwY can be spliced in membrane 1 by rules 7 and 8.

If $(\mid Z'_\lambda, X \mid wY) \vdash_7 (wY, X Z'_\lambda)$ is performed, the first output string, sent out of the system, does not contribute to the language generated by Π as $Y \notin T$; the second output string remains in membrane 1 and cannot be involved in any splicing.

If $w = xB, x \in \{N \cup T\}^*$ then $(Xx \mid BY, Z_\lambda \mid) \vdash_6 (Xx, Z_\lambda BY)$ can be performed. The first output string, remaining in the same membrane, can be involved in $(\mid Z'_\lambda, X \mid x) \vdash_7 (x, X Z'_\lambda)$. The strings $x, Z_\lambda BY$ and $X Z'_\lambda$ are sent out of the system but only x can contribute to the language generated by Π.

If $x \in T^*$ the system Π has simulated a derivation of G.

In the initial configuration of membrane 2 no splicing can be performed.

As just demonstrated all derivations in G can be simulated in Π and, conversely, all correct computations in Π correspond to correct derivations in G. As we only collect strings in T^* leaving the system Π, we have $L(G) = L(\Pi)$ proving the theorem.

Considering the definitions (2) and (4) it is easy to see that $SPL(i/o, 2, 2, (2, 1, 1, 2)) = RE$. The proof is similar to the one of Theorem 1 where for each rule the target indications are switched.

4 P systems on asymmetric graphs

By $SP'L(go, *)$ we denote the union of all families $SP'L(go, m), m, m \leq 1$; in [11] the authors demonstrate that $SP'L(go, *) = RE$ (Theorem 9). Hereby we improve this result demonstrating that $SP'L(go, 3) = RE$ and, considering that $SP'L(go, 1) = SP'L(go, 2) = REG$ (Theorem 7 in [11]), our result is minimal.

A simple way to prove that $SP'L(go, 3, (1, 2, 2, 1)) = SP'L(go, 3, (2, 1, 1, 2)) = RE$ is using Theorem 1. If we consider the graph and the planar map represented in Figure 1 we can imagine that membranes 1 and 2 have the same languages and similar set of evolution rules of membranes 1 and 2 (respectively) present in Theorem 1. Membrane 3 is only used to pass strings from membrane 2 to membrane 1 without changing them. Each splicing rule present in Theorem 1 and containing *in* as target indication is present in the P system on asymmetric graph with *go* instead of *in*, the other target indications are not changed.

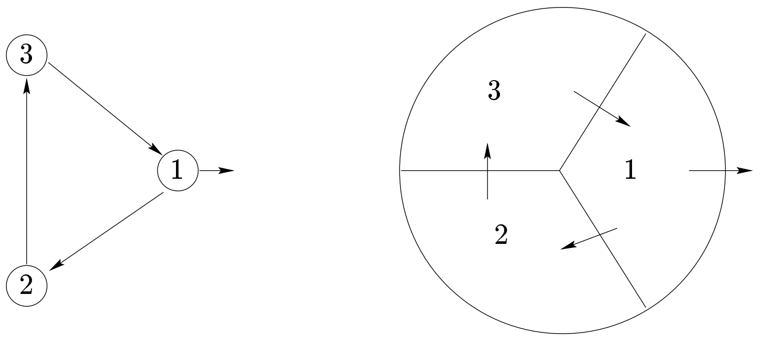


Figure 1: Graph system and planar map in the proof of Theorem 2

The language associated with membrane 3 is $\{Z\}$ and the set of evolution rules is $\{13)(\alpha\#\$Z\#; go, here) \mid \alpha \in \{Y, Y_1, Y_2, Z_{X_i} \mid 1 \leq i \leq n + m\}\}$. The passage of strings from membrane 2 to membrane 1 is made through membrane 3: the first output string is sent to membrane 1, the second, Z, remaining in membrane 3, belongs to its language. No splicing is possible in the initial configuration of membrane 3.

Keeping the number of membranes equal to 3 it is possible to reduce the diameter of a P system on asymmetric graph generating RE.

Theorem 2 $SP'L(go, 3, (0, 2, 1, 0)) = SP'L(go, 3, (1, 0, 0, 2)) = RE$.

Proof. We only prove that $SP'L(go, 3, (0, 2, 1, 0)) = RE$, the other equality can be obtained using this proof and definitions (2) and (4).

Let $G = (N, T, S, R)$ be a type-0 Chomsky grammar in Kuroda normal form (this means that the productions in R can be of the form $A \to a, A \to CD, AC \to DE$ or $A \to \lambda$ where $A, C, D, E \in N$ and $a \in T$) and B be a symbol not in $N \cup T$. Let us assume that symbols in $N \cup T \cup \{B\}$ can be numbered in a one-to-one manner so that $N \cup T \cup \{B\} = \{\alpha_1, \cdots, \alpha_n\}$ and that R contains m productions: $u_i \to v_i, 1 \leq i \leq m$. Moreover R can be divided in two sets: $R_1 = \{u_i \to v_i \mid u_i \to v_i \in R \ \wedge |u_i| = 1\}$ and $R_2 = \{u_i \to v_i \mid u_i \to v_i \in R \ \wedge |u_i| = 2\}$ so that $R_1 \cup R_2 = R$ and $R_1 \cap R_2 = \emptyset$. Consider also $R' = \{u \to u \mid u \in \{\alpha_1, \cdots, \alpha_n\}\}$ and that $\{X, X', Y, Y', Z_X, Z_{X'}, Z_Y, Z_{Y'}, Z_\lambda, Z'_\lambda\} \cup \{X_i, Y_i, Z_i, Z_{X_i}, Z_{Y_i} \mid 1 \leq i \leq n+m\} \cup \{Y'_i, Z_{Y'_i} \mid u_i \to v_i \in R_2\}$ are symbols not in $N \cup T$.

Hereby the P system on asymmetric graph of degree 3 and diameter (0, 2, 1, 0) simulating the just defined grammar is described. For a better understanding of the demonstration splicing rules are numbered.

$$
\begin{aligned}
\Pi = & \{V, T, g, L_1, L_2, L_3, R_1, R_2, R_3\}, \\
V = & N \cup T \cup \{B, X, X', Y, Y', Z_X, Z_{X'}, Z_Y, Z_{Y'}, Z_\lambda, Z'_\lambda\} \cup \\
& \{X_i, Y_i, Z_i, Z_{X_i}, Z_{Y_i} \mid 1 \leq i \leq n+m\} \cup \{Y'_i, Z_{Y'_i} \mid u_i \to v_i \in R_2\}, \\
g = & \{(1,2), (2,3), (3,1)\}, \\
L_1 = & \{XBSY, X'Z_{X'}, Z_\lambda, Z'_\lambda\} \cup \{Z_{Y_i}Y_i \mid 1 \leq i \leq n+m\} \cup \\
& \{X_iZ_{X_i} \mid 1 \leq i \leq n+m-1\} \cup \{Z_{Y'_i}Y'_i \mid u_i \to v_i \in R_2\}, \\
L_2 = & \{Z_{Y'}Y'\} \cup \{X_iv_iZ_i \mid 1 \leq i \leq n+m\} \cup \{Z_{Y_i}Y_i \mid 1 \leq i \leq n+m-1\}, \\
L_3 = & \{XZ_X, Z_YY\} \cup \{X_iZ_{X_i} \mid 2 \leq i \leq n+m\}, \\
R_1 = & \{1)(\#u_iY\$Z_{Y_i}\#; go, out) \mid 1 \leq i \leq n+m\} \cup \\
& \{2)(\#CY\$Z_{Y'_i}\#; here, out), 3)(\#AY'_i\$Z_{Y_i}\#; go, out) \mid u_i \to v_i \in R_2\} \cup \\
& \{4)(\#Z_{X_{i-1}}\$X_i\#; go, out) \mid 2 \leq i \leq n+m\} \cup \\
& \{5)(\#Z_{X'}\$X_1\#; go, out), 6)(\#BY\$Z_\lambda\#; here, out), 7)(\#Z'_\lambda\$X\#; out, out)\}, \\
R_2 = & \{8)(\#Z_i\$X\#; go, go) \mid 1 \leq i \leq n+m\} \cup \\
& \{9)(\#Y_i\$Z_{Y_{i-1}}\#; go, go) \mid 2 \leq i \leq n+m\} \cup \{10)(\#Y_1\$Z_{Y'}\#; go, go)\}, \\
R_3 = & \{11)(\#Z_{X_i}\$X_i\#; go, here) \mid 2 \leq i \leq n+m\} \cup \\
& \{12)(\#Z_X\$X'\#; go, go), 13)(\#Y'\$Z_Y\#, here, go\}
\end{aligned}
$$

The idea of the proof is again based on the "rotate-and-simulate" technique. The sentential forms generated by G are simulated in Π in a circular permutation $Xw_1Bw_2Y, w_1, w_2 \in \{N \cup T\}^*$, with variants of X and Y. They will be present in a membrane of Π if and only if w_2w_1 is a sentential form of G. It is possible to remove the nonterminal symbol Y only with B from strings of the form $XwBY$. In this way the correct permutation of the string is ensured.

The simulation of a production in R and the rotation are done in the same way.

Assume that in membrane 1 we have a string of the form Xwu_iY with $w, u_i \in \{N \cup T \cup \{B\}\}^*$ (initially we have $XBSY$).

If a production in $R_1 \cup R'$ is simulated we have $(Xw \mid u_iY, Z_{Y_i} \mid Y_i) \vdash_1 (XwY_i, Z_{Y_i}u_iY)$ the first output string is sent into membrane 2 while the second is sent out of the system.

If a production in R_2 is simulated we have $(XwA \mid CY, Z_{Y'_i} \mid Y'_i) \vdash_2 (XwAY'_i, Z_{Y'_i}CY)$ (the first output string remains in membrane 1 and the second leaves the system) and then $(Xw \mid AY'_i, Z_{Y_i} \mid Y_i) \vdash_3 (XwY_i, Z_{Y_i}AY'_i)$ (the first output string is sent to membrane 2 and the second leaves the system).

In both cases the suffix u_iY is changed with $Y_i, 1 \leq i \leq n+m$. The strings leaving the system do not belong to T^* so they do not contribute to the language generated by Π.

In membrane 2, with a string as XwY_i, it is possible to perform $(X_jv_j \mid Z_j, X \mid wY_i) \vdash_8 (X_jv_jwY_i, XZ_j)$ (for some $1 \leq j \leq n+m$), and both strings are sent to membrane 3, where only the first can be involved in splicing operations.

A string as $X_jv_jwY_i$ is spliced so to decrease the value of the subscripts of X and Y until special situations are present. The subscript of Y is decreased in membrane 1, the one of X in membrane 2; membrane 3 is simply used to pass strings during this process.

So when a string of the form $X_jv_jwY_i, 2 \leq j \leq n+m$ is present in membrane 3 it is moved to membrane 1 by $(X_j \mid Z_{X_j}, X_j \mid v_jwY_i) \vdash_{11} (X_jv_jwY_i, X_jZ_{X_j})$. The string $X_jZ_{X_j}$, remaining in membrane 3, belongs to its language.

In membrane 1 it is possible to have $(X_{j-1} \mid Z_{j-1}, X_j \mid v_jwY_i) \vdash_4 (X_{j-1}v_jwY_i, X_jZ_{j-1})$. The first output string is sent to membrane 2, the second leaves the system (but do not contributes to the language generated by Π as it is not in T^*).

A string as $X_{j-1}v_jwY_i$ can be spliced in membrane 2 so to have $(X_{j-i}v_jw \mid Y_i, Z_{i-1} \mid Y_{i-1}) \vdash_9 (X_{j-i}v_jwY_{i-1}, Z_{i-1}Y_i)$, both output strings are sent to membrane 3 but the second one cannot be involved in any splicing.

Decreasing the subscripts of X and Y it is possible to have: $X_1v_jwY_k$ in membrane 1, $X_kv_jwY_1$ in membrane 2 or $X'v_jwY'$ in membrane 3, where $2 \leq k \leq n+m$.

In the first case $(X' \mid Z_{X'}, X_1 \mid v_jwY_k) \vdash_5 (X'v_jwY_k, X_1Z_{X'}))$ is performed. The string $X_1Z_{X'}$ is sent out of the system and do not contributes to the language generated by Π as it is not in T^*. The first output string ins sent to membrane 2 where the subscript of Y is decreased so to have $X'v_jwY_{k-1}$ which is sent to membrane 3. Here X' is substituted with X by $(X \mid Z_X, X' \mid v_jwY_{k-1}) \vdash_{12} (Xv_jwY_{k-1}, X'Z_X)$. Both strings are sent to membrane 1 and no splicing can be performed on them.

In the second case the Y_1 in $X_kv_jwY_1$ is substituted with Y' in membrane 2 by $(X_kv_jw \mid Y_1, Z_{Y'} \mid Y') \vdash_{10} (X_kv_jwY', Z_{Y'}Y_1)$ and both output strings are sent to membrane 3. Here only the first one can be involved in a splicing operation changing Y' in Y: $(X_kv_jw \mid Y', Z_Y \mid Y) \vdash_{13} (X_kv_jwY, Z_YY')$. The first output string remains in membrane 3, the second is sent to membrane 1. In both cases no splicing can be performed on them.

In the third case two directions of splicing are possible. If $(X \mid Z_X, X' \mid v_jwY') \vdash_{12} (Xv_jwY', X'Z_X)$ is performed the two output strings are sent to membrane 1 where no splicing rule can be applied on them. If $(X'v_jw \mid Y', Z_Y \mid Y) \vdash_{13} (X'v_jwY, Z_YY')$ is performed the second output string is sent to membrane 1 where no splicing can be performed on it. The string $X'v_jwY$ remains in membrane 3 where X' can be changed with X by rule 12 so to obtain Xv_jwY and $X'Z_X$. both sent to membrane 1. Here the string $X'Z_X$ cannot be involved in any splicing.

What just described is the process to pass from Xwu_iY to Xv_jwY simulating a production in R or rotating the substring between X and Y of one symbol.

At any moment a string of the form XwY can be spliced in membrane 1 by rules 6 and 7.

If $(| \, Z'_\lambda, X \mid wY) \vdash_7 (wY, XZ'_\lambda)$ is performed the first output string, sent out of the system, does not contribute to the language generated by Π as $Y \notin T$; the second output string remains in membrane 1 and cannot be involved in any splicing.

If $w = xB, x \in \{N \cup T\}^*$ then $(Xx \mid BY, Z_\lambda \, |) \vdash_6 (Xx, Z_\lambda BY)$ can be performed. The first output string, remaining in the same membrane, can be involved in $(| \, Z'_\lambda, X \mid x) \vdash_7 (x, XZ'_\lambda)$. The strings $x, Z_\lambda BY$ and XZ'_λ are sent out of the system but only x can contribute to the language generated by Π.

If $x \in T^*$, the system Π has simulated a derivation of G.

If we consider the three membranes in their initial configurations we can see that the splicing operations that can be performed do not produce any terminal string.

In membrane 1 it is possible to have $(X_{i-1} \mid Z_{X_{i-1}}, X_i \mid Z_{X_i}) \vdash_4 (X_{i-1}Z_{X_i}, X_iZ_{X_{i-1}})$ and $(X' \mid Z_{X'}, X_1 \mid Z_{X_1})$. In both cases the first output strings are sent to membrane 2 where no splicing can be performed; the second exit the system but do not contribute to the language generated by Π as not terminal.

In membrane 2 the splicing operation $(Z_{Y_i} \mid Y_i, Z_{Y_{i-1}} \mid Y_{i-1}) \vdash_9 (Z_{Y_i}Y_{i-1}, Z_{Y_{i-1}}Y_i)$ generates two strings sent to membrane 3 and no longer used.

In membrane 3 it is possible to have $(X_i \mid Z_{X_i}, X_i \mid Z_{X_i}) \vdash_{11} (X_iZ_{X_i}, X_iZ_{X_i})$. The first output string is sent to membrane 1 while the second, remaining in membrane 3, belongs to its alphabet. In membrane 1 the use of the rule 4 brings to $(X_{i-1} \mid Z_{i-1}, X_i \mid Z_{X_i}) \vdash_4 (X_{i-1}Z_{X_i}, X_iZ_{i-1})$. The first output string is sent to membrane 2 and no longer used; the second exit the system but do not contribute to the language generated by Π as not terminal.

As just demonstrated all derivations in G can be simulated in Π and, conversely, all correct computations in Π correspond to correct derivations in G. As we only collect strings in T^* leaving the system Π, we have $L(G) = L(\Pi)$ proving the theorem.

5 Final remarks

We have considered P systems based on splicing having a tree or a graph as structure. In both cases improvements of theorems demonstrating their generative capability were found. In particular our result concerning splicing P systems on asymmetric graphs is minimal.

Acknowledgments

I thank the Universitá degli Studi di Milano for its financial support to my PhD and the Universiteit Leiden, personified by Prof. G. Rozenberg, accepting me as PhD student in his friendly group of research.

References

[1] K. Culik II, T. Harju, Splicing semigroups of dominoes and DNA, *Discrete Appl. Math.*, 31 (1991), 261-277.

[2] T. Head, Formal language theory and DNA; an analysis of the generative capacity of specific recombinant behaviors, *Bull. Math. Biology*, 49 (1987), 737 - 759.

[3] A. Păun, Controlled H systems of small radius, *Fundamenta Informaticae*, 31, 2 (1997), 185 - 193.

[4] A. Păun, M. Păun, On the membrane computing based on splicing, submitted, 2000

[5] Gh. Păun, Computing with membranes. *Journal of Computer and System Sciences*, 61 (2000), and also Turku Centre for Computer Science-TUCS Report No. 208, 1998 http://www.tucs.fi.

[6] Gh. Păun, Computing with membranes. An introduction, *Bulletin of the EATCS*, 67 (Febr. 1999), 139-152.

[7] Gh. Păun, Computing with membranes - A variant: P systems with polarized membranes, *Inter. J. of Foundations of Computer Science*, 11, 1 (2000), 167-182, and *Auckland Univ. CDMTCS Report* No. 089, 1999, http://www.cs.auckland.ac.nz/CDMTCS.

[8] Gh. Păun, Regular extended H systems are computationally universal, *J. Automata, Languages, Combinatorics*, 1, 1 (1996), 27 - 36.

[9] Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing.* New Computing Paradigms, Springer-Verlag, Berlin, 1998.

[10] Gh. Păun, Y. Sakakibara, T. Yokomori. P systems on graphs of restricted forms, submitted, 1999.

[11] Gh. Păun, T. Yokomori, Membrane computing based on splicing. In E. Winfree and D. Gifford,
editors, *DNA Based Computers V.* MIT, June 1999,
http://bramble.princeton.edu/DNA5/Tarfiles/paun.tgz. Article accepted to the DIMACS 5^{th} International Meeting on DNA Based Computers.

[12] D. Pixton, Regularity of splicing languages, *Discrete Appl. Math.*, 69 (1996), 101-124

Concentration Prediction of Pattern Reaction Systems

Satoshi Kobayashi
Dept. of Information Sciences, Tokyo Denki University
Ishizaka, Hatoyama-machi, Hiki-gun, Saitama 350-0394, JAPAN
e-mail:satoshi@j.dendai.ac.jp

Abstract

In this paper, we will propose a formal system for analyzing the computational capability of chemical reaction systems of linear molecules. In this model, each linear molecule is represented as a string w with a real value c, where c is the concentration of the molecule w. Thus, the system could be regarded as a real-valued multiset system dealing with linear structures (strings). We further discuss on the problem of predicting the concentration of a molecule w at the specified time t in a given chemical reaction system. In particular, we give a polynomial time prediction algorithm for ligation reaction systems.

1 Introduction

Since Adleman's seminal paper on a DNA solution to directed Hamiltonian path problem ([Adl94]), there have been proposed many models of DNA computation, based on string manipulations ([Adl96][Win96]), nondeterministic Turing machines ([Rei95][Rot96]), boolean circuit ([OR98]), splicing operations ([Hea87][PRS99]), horn clause computation ([KYSM97][Mih97][Kob99]), etc. Although these works presented some interesting aspects of computational capability of chemical reactions, from a realistic point of views, there exists a problem that the concentration of each molecule is not considered in their models.

In this paper, we will propose a computational model of chemical reaction systems with linear molecules, in which every molecule has its concentration. In this model, each linear molecule is represented as a string w with a real value c, where c is the concentration of the molecule w. Thus, the system could be regarded as a real-valued multiset system dealing with linear structures (strings). We further discuss on the problem of predicting the concentration of a molecule w at the specified time t in a given chemical reaction system. In particular, we give a polynomial time prediction algorithm for ligation reaction systems.

Inspired from the information processing by biological molecules in a cell, Păun proposed a parallel computation model, *computing with membranes* (*P-system*), in which contents of a cell is represented as a multiset of objects (molecules), and discusses on the computational capability of the multiset processing with membrane structures. Further, the model is extended in order to deal with linear molecules ([KR99][Pau00][PP00][Pau98][PY99] [Zan00a][Zan00b]). Although most of these works use the concentration model which assigns an integer to each molecule, the current paper assumes that each molecule has a real value as its concentration. Furthermore, we have interests in analyzing the computational capability of real valued dynamical systems, which is approximately obtained from differential equation systems representing the kinetics of chemical reactions.

Hagiya and Nishikawa ([HN99]) proposed a model of molecular computation motivated from the work by Berry and Boudol ([BB92]). They claim that it is important to deal with in the model (1) the concentration of each molecule, and (2) the rate of each chemical reaction. The reactions of their model is classified into three basic types: assembly, dissociation, and state transition. One of their open research topics include a problem of analyzing the computational capability of the system with various restrictions on the reaction types. In particular, they have interests in the relationship between the computational capability and molecular topologies, or in the effect of simultaneous state transitions on the computational capability of the systems.

The purpose of the present work is to give a first step toward answering one of such questions, i.e. revealing the computational capability of chemical reactions of linear molecules. For that purpose, we will propose a realistic model of molecular computation, inspired from differential equations representing chemical reactions. Although the proposed system is discretized in time and thus cannot deal with actual chemical reactions, we think that the proposed model could be used as an approximation of real chemical reaction systems.

In section 2, we propose our model of molecular computation and its relationship to actual chemical reactions. Section 3 describes an efficient algorithm for predicting the concentration of a given molecule at the specified discrete time in a given ligation system. This result suggests that the ligation reaction of linear molecules does not have computational capability beyond the class P ([GJ79]), even if we consider the concentration of molecules. Conclusions and open research topics are given in section 4.

2 Pattern Reaction System: A Model of Chemical Reaction System

Let Σ be a finite alphabet, V be a countable set of variables, and $\mathcal{F}$ be a countable set of function symbols such that each element f of $\mathcal{F}$ is associated with a function $\hat{f} : \Sigma^* \to \Sigma^*$. The length of a string $w \in \Sigma^*$ is denoted by $\mid w \mid$. By $\mathcal{F}(V)$, we denote the set $\{f(X) \mid f \in \mathcal{F}, X \in V\}$. We can regard V and $\mathcal{F}(V)$ as countable alphabets. Thus, in the sequel, we often regard elements $X \in V$ and $f(X) \in \mathcal{F}(V)$ as single letters. An *f-pattern* is a non-empty string over $\Sigma \cup V \cup \mathcal{F}(V)$. For a pattern p, by $\mid p \mid$, we denote the length of p as a string.

A *ground substitution* (or *substitution*, in short) θ is a mapping from V to Σ^*.

For an f-pattern p and a ground substitution θ, we define:

$$p\theta \quad \equiv_{def} \begin{cases} \theta(X) & \text{if } p \text{ is a variable } X \\ c & \text{if } p \text{ is a symbol } c \in \Sigma \\ \hat{f}(\theta(X)) & \text{if } p \text{ is of the form } f(X) \text{ for some } X \in V \\ p_1\theta \cdot p_2\theta & \text{if } p \text{ is of the form } p_1p_2 \text{ for some f-patterns } p_1, p_2 \end{cases}$$

A rule of the form $r : q_1, ..., q_m \leftarrow p_1, ..., p_n$, where p_i $(i = 1, ..., n)$ and q_i $(i = 1, ..., m)$ are f-patterns, is called a *reaction rule*. The size $size(r)$ of r is defined as $\sum_{i=1}^{m} \mid q_i \mid + \sum_{i=1}^{n} \mid p_i \mid$. The f-pattern q_i $(i = 1, ..., m)$ is called a *product* of r, and the f-pattern p_i $(i = 1, ..., n)$ is called a *resource* of r. By $V(r)$, we denote the set of all variables appearing in the rule r. In this paper, we assume that each reaction rule r is associated with a function f_r from $\mathbf{R}^n$ to $\mathbf{R}$, where n is the number of resources of r and $\mathbf{R}$ is the set of real numbers. A finite subset of reaction rules is called a *pattern reaction system* (PRS). For a PRS P, by $size(P)$, we denote $\sum_{r \in P} size(r)$. By $\mathcal{PRS}$, we denote the set of all PRSs.

Example 1 Let $\Sigma = \{a, c, g, t, [a/t], [c/g], [g/c], [t/a]\}$ and consider two function symbols f_1, f_2 whose associated functions are defined as follows:

$$\begin{array}{l} \hat{f}_1(a) = t,\ \hat{f}_1(c) = g,\ \hat{f}_1(g) = c,\ \hat{f}_1(t) = a, \\ \hat{f}_1(x \cdot w) = \hat{f}_1(w)\hat{f}_1(x) \quad \text{for } x \in \{a, c, g, t\},\ w \in \{a, c, g, t\}^*, \\ \hat{f}_2(a) = [a/t],\ \hat{f}_2(c) = [c/g],\ \hat{f}_2(g) = [g/c],\ \hat{f}_2(t) = [t/a], \\ \hat{f}_2(w_1w_2) = \hat{f}_2(w_1)\hat{f}_2(w_2), \quad \text{for } w_1, w_2 \in \{a, c, g, t\}^*. \end{array}$$

Then, the complete hybridization of two DNA molecules based on Watson-Crick complementarity can be represented by the following reaction rule:

$$f_2(X) \leftarrow X, f_1(X).$$

□

The pattern reaction system has a close relation to the elementary formal system (EFS), whose computational capability and learnability from positive data are well studied([Smu61][ASY92][Shi94]). However, PRS is different from EFS in that it deals with a real valued multiset.

Let us consider the following two chemical reactions:

$$\mathrm{A}_1 + \mathrm{A}_2 \overset{k_1}{\rightarrow} \mathrm{A}_4,$$

$$\mathrm{A}_1 + \mathrm{A}_3 \overset{k_2}{\rightarrow} \mathrm{A}_5,$$

where k_1 and k_2 are the rate constants of the above reactions. Differential equations to model these chemical reactions can be written as follows:

$$\begin{aligned} \frac{d[\mathrm{A}]_1}{dt} &= -k_1[\mathrm{A}]_1[\mathrm{A}]_2 - k_2[\mathrm{A}]_1[\mathrm{A}]_3, \\ \frac{d[\mathrm{A}]_2}{dt} &= -k_1[\mathrm{A}]_1[\mathrm{A}]_2, \end{aligned}$$

$$
\begin{aligned}
\frac{d[\mathrm{A}]_3}{dt} &= -k_2[\mathrm{A}]_1[\mathrm{A}]_3,\\
\frac{d[\mathrm{A}]_4}{dt} &= k_1[\mathrm{A}]_1[\mathrm{A}]_2,\\
\frac{d[\mathrm{A}]_5}{dt} &= k_2[\mathrm{A}]_1[\mathrm{A}]_3.
\end{aligned}
$$

Let us denote by $[\mathrm{A}]_i(t)$ the concentration of the molecule $[\mathrm{A}]_i$ at time t. Then, a naive numerical calculation gives the values $[\mathrm{A}]_i(t+\Delta t)$ for small Δt as follows:

$$
\begin{aligned}
[\mathrm{A}]_1(t+\Delta t) &= [\mathrm{A}]_1(t) - k_1[\mathrm{A}]_1[\mathrm{A}]_2\Delta t - k_2[\mathrm{A}]_1[\mathrm{A}]_3\Delta t,\\
[\mathrm{A}]_2(t+\Delta t) &= [\mathrm{A}]_2(t) - k_1[\mathrm{A}]_1[\mathrm{A}]_2\Delta t,\\
[\mathrm{A}]_3(t+\Delta t) &= [\mathrm{A}]_3(t) - k_2[\mathrm{A}]_1[\mathrm{A}]_3\Delta t,\\
[\mathrm{A}]_4(t+\Delta t) &= [\mathrm{A}]_4(t) + k_1[\mathrm{A}]_1[\mathrm{A}]_2\Delta t,\\
[\mathrm{A}]_5(t+\Delta t) &= [\mathrm{A}]_5(t) + k_2[\mathrm{A}]_1[\mathrm{A}]_3\Delta t.
\end{aligned}
$$

Inspired from this naive method for calculating the concentrations of the molecules, we will propose bellow a dynamics of PRS.

Let X be any set. A function from X to $\mathbf{R}$ is called a *real valued multiset* (or *multiset*, for short) over X. The value $M(x)$ of an object $x \in X$ represents the concentration of x. By $supp(M)$, we denote the set $\{x \mid M(x) \neq 0\}$. We say that a multiset M is *finite* if $supp(M)$ is finite. For any finite relation M from X to $\mathbf{R}$, i.e. $M \subseteq X \times \mathbf{R}$, by $\Gamma(M)$, we denote a function from X to $\mathbf{R}$ defined as:

$$
\Gamma(M)(x) = \sum_{(x,v)\in M} v, \qquad \text{for every } x \in X.
$$

Note that we sometimes regard a multiset M as a relation $M \subseteq X \times \mathbf{R}$. For any finite relation M from X to $\mathbf{R}$, by $size(M)$, we denote $\sum_{(w,c)\in supp(M)} \mid w \mid$.

For a pattern reaction system P and a multiset M over Σ^*, we define:

$$
\begin{aligned}
\delta_P(M) \;= \{ \quad & (q_1\theta, v), ..., (q_m\theta, v), (p_1\theta, -v), ..., (p_n\theta, -v) \quad \mid \\
& r : q_1, ..., q_m \leftarrow p_1, ..., p_n \;\in\; P,\\
& \theta \text{ is a ground substitution defined only on } V(r),\\
& v = f_r(M(p_1\theta), ..., M(p_n\theta)) \quad \},\\
\gamma_P(M) \;=\; & \Gamma(M \cup \delta_P(M)),\\
\gamma_P^0(M) \;=\; & M,\\
\gamma_P^i(M) \;=\; & \gamma_P(\gamma_P^{i-1}(M)), \quad \text{for every } i \geq 1.
\end{aligned}
$$

Thus, the pattern reaction system could be regarded as a dynamical system which transforms a multiset over Σ^*.

Now we will describe bellow how to use this system as a computational device for solving decision problems. Let A be an alphabet, and Q be a decision problem defined as a function from A^* to $\{0, 1\}$. For a problem instance $w \in A^*$, $Q(w)$ is the answer to the question w.

Let $\mathbf{N}$ be the set of nonnegative integers, and $\mathcal{FM}$ be the set of all finite multisets over Σ^*. An *encoding function* is a function from A^* to $\mathcal{FM}$ which can be computed in polynomial time. A *PRS generator* is a function from A^* to $\mathcal{PRS}$ which can be computed in polynomial time. A time function is a function from A^* to $\mathbf{N}$ which can be computed in polynomial time.

We say that a decision problem Q can be *computed in polynomial steps using PRS* if there exist an encoding function α, a PRS generator β, a time function T, a real value $h \in \mathbf{R}$, and a string $w_g \in \Sigma^*$ such that for every problem instance $w \in A^*$, $\gamma_{\beta(w)}^{T(w)}(\alpha(w))(w_g) \geq h$ holds if and only if $Q(w) = 1$. In this definition, h and w_g are called a *threshold* and a *goal molecule*, respectively.

In the next section, we consider the following problem:

[**Concentration Prediction Problem(CPP)**]

Input: a PRS P, a finite multiset M over Σ^* and an integer $t > 0$.

Output: the value $\gamma_P^t(M)(w)$.

We say that CPP for a subclass $\mathcal{P}$ of $\mathcal{PRS}$ is *efficiently computable* if there exists an algorithm which for every $P \in \mathcal{P}$, a multiset M over Σ^* and an integer $t > 0$ computes the value $\gamma_P^t(M)(w)$ in polynomial time with respect to $size(P)$, $size(M)$, t and $\mid w \mid$. Note that in this paper we assume that basic operations of real values, such as addition and multiplication, could be computed in a constant time.

The problem CPP is closely related to the computational capability of a PRS, which is shown in the following theorem:

Theorem 1 Assume that CPP for a subclass $\mathcal{P}$ of $\mathcal{PRS}$ is efficiently computable, and that a decision problem Q can be computed in polynomial steps using PRS with a PRS generator β such that β only produces elements of $\mathcal{P}$. Then, Q can be computed in polynomial time by deterministic Turing machines.

Proof

For a problem instance $w \in A^*$, we execute the efficient algorithm A_{CPP} for CPP with inputs of the PRS $\beta(w)$, the multiset $\alpha(w)$ and the integer $T(w)$. We return the value 1 if and only if the answer from A_{CPP} is greater than or equal to the threshold h. This algorithm computes the solution for $Q(w)$ and runs in polynomial time. □

3 Concentration Prediction of Ligation Systems

For a string w over Σ, by $prf_k(w)$ and $suf_k(w)$, we denote the prefix and the suffix of w of length k, respectively. In case of $\mid w \mid < k$, both of $prf_k(w)$ and $suf_k(w)$ are not defined. For a set L of strings, by $Prf_k(L)$ and $Suf_k(L)$, we denote the set $\{prf_k(w) \mid w \in L\}$ and $\{suf_k(w) \mid w \in L\}$, respectively.

A *simple ligation system* is a PRS consisting of reaction rules of the form: $Xw_1w_2Y \leftarrow Xw_1, w_2Y$ which is associated with a function $f_r(x, y) = k_r xy$, where k_r is called the *rate constant* of the reaction r. For a rule r of the form $Xw_1w_2Y \leftarrow Xw_1, w_2Y$, $max(\mid w_1 \mid, \mid w_2 \mid)$ is called the *radius* of r.

Let P be a simple ligation system, and k be the maximum of the radius of rules in P.

In the sequel, we will assume that the input multiset M of the concentration prediction problem should satisfy the following condition:

$\mid w \mid \geq k$ holds for every $w \in supp(M)$.

It is clear that the following proposition holds:

Proposition 1 For any input M of multiset over Σ^* and a simple ligation system P satisfying the above condition, the following equations hold for every $t \geq 0$:
$Prf_k(supp(\gamma_P^t(M)) \subseteq Prf_k(supp(M)), \quad Suf_k(supp(\gamma_P^t(M)) \subseteq Suf_k(supp(M)).$ $\square$

Let M be a finite multiset over Σ^* satisfying the condition above. For a reaction rule $r : Xw_1w_2Y \leftarrow Xw_1, w_2Y$ in P and a pair (u, v) of strings, we write $r \rightarrow (u, v)$ if and only if w_1 is a suffix of u and w_2 is a prefix of v. Note that $r \rightarrow (u, v)$ holds if and only if there exists a ground substitution θ such that $Xw_1\theta = u$ and $w_2Y\theta = v$.

For every integer $t \in \mathbf{N}$, we define a multiset $C(t)$ over $Prf_k(\Sigma^*) \times Suf_k(\Sigma^*)$ inductively as follows:

$$\begin{array}{rcl}
C(0) & = & \Gamma(\{((prf_k(w), suf_k(w)), M(w)) \mid w \in \Sigma^*\}), \\
C(t+1) & = & \Gamma(C(t) \cup \delta C_1(t) \cup \delta C_2(t) \cup \delta C_3(t)), \quad (t \geq 0) \\
\delta C_1(t) & = & \{((p,q), C(t)((p,u)) \cdot C(t)((v,q)) \cdot k_r) \mid p,q,u,v \in \Sigma^k, r \in P, r \rightarrow (u,v)\}, \\
\delta C_2(t) & = & \{((p,q), -C(t)((p,q)) \cdot C(t)((u,v)) \cdot k_r) \mid p,q,u,v \in \Sigma^k, r \in P, r \rightarrow (q,u)\}, \\
\delta C_3(t) & = & \{((p,q), -C(t)((u,v)) \cdot C(t)((p,q)) \cdot k_r) \mid p,q,u,v \in \Sigma^k, r \in P, r \rightarrow (v,p)\}.
\end{array}$$

We have the following lemma:

Lemma 1 For every $t \geq 0$, the following equation holds:

$$C(t) \quad = \quad \Gamma(\{((prf_k(w), suf_k(w)), \gamma_P^t(M)(w)) \mid w \in \Sigma^*\}).$$

Proof
We will prove the claim by induction on $t \geq 0$.

In case of $t = 0$, the definition of $C(0)$ gives the claim. Assume that the claim holds for the case of $t \leq i$ and let

$$R = \Gamma(\{((prf_k(w), suf_k(w)), \gamma_P^{i+1}(M)(w)) \mid w \in \Sigma^*\}).$$

Then, we have:

$$\begin{array}{rcl}
R & = & \Gamma(\{((p,q), \Gamma(\gamma_P^i(M) \cup \delta_P(\gamma_P^i(M)))(w)) \mid w \in \Sigma^*, \ p,q \in \Sigma^k, \\
& & \qquad\qquad p = prf_k(w), \ q = suf_k(w)\}) \\
& = & \Gamma(\ \{((p,q), \gamma_P^i(M)(w)) \mid w \in \Sigma^*, \ p,q \in \Sigma^k, \\
& & \qquad\qquad p = prf_k(w), \ q = suf_k(w)\} \quad \cup \\
& & \quad \{((p,q), \Gamma(\delta_P(\gamma_P^i(M)))(w)) \mid w \in \Sigma^*, \ p,q \in \Sigma^k,
\end{array}$$

$$\begin{array}{rl}
& \qquad\qquad\qquad\qquad p = prf_k(w),\ q = suf_k(w)\} \ \ ) \\
= & \Gamma(\ \ C(i)\ \cup\ \{((p,q),\Gamma(\delta_P(\gamma_P^i(M)))(w))\ \mid\ w\in\Sigma^*,\ p,q\in\Sigma^k, \\
& \qquad\qquad\qquad\qquad p = prf_k(w),\ q = suf_k(w)\} \ \ ) \\
= & \Gamma(\ \ C(i)\ \cup\ X_1 \cup\ X_2 \cup\ X_3\ \ ),
\end{array}$$

where

$$\begin{array}{rcl}
X_1 & = & \Gamma(\ \{\ ((p,q),c)\ \mid\ r\in P,\ p,q,u,v\in\Sigma^k,\ w_1,w_2\in\Sigma^*,\ prf_k(w_1)=p, \\
& & \qquad suf_k(w_1)=u,\ prf_k(w_2)=v,\ suf_k(w_2)=q, \\
& & \qquad r\to(u,v),\ c = k_r\cdot\gamma_P^i(M)(w_1)\cdot\gamma_P^i(M)(w_2)\ \}\ ), \\
X_2 & = & \Gamma(\ \{\ ((p,q),c)\ \mid\ r\in P,\ p,q,u,v\in\Sigma^k,\ w_1,w_2\in\Sigma^*,\ prf_k(w_1)=p, \\
& & \qquad suf_k(w_1)=q,\ prf_k(w_2)=u,\ suf_k(w_2)=v, \\
& & \qquad r\to(q,u),\ c = -k_r\cdot\gamma_P^i(M)(w_1)\cdot\gamma_P^i(M)(w_2)\ \}\ ), \\
X_3 & = & \Gamma(\ \{\ ((p,q),c)\ \mid\ r\in P,\ p,q,u,v\in\Sigma^k,\ w_1,w_2\in\Sigma^*,\ prf_k(w_1)=u, \\
& & \qquad suf_k(w_1)=v,\ prf_k(w_2)=p,\ suf_k(w_2)=q, \\
& & \qquad r\to(v,p),\ c = -k_r\cdot\gamma_P^i(M)(w_1)\cdot\gamma_P^i(M)(w_2)\ \}\ ).
\end{array}$$

Then, we can obtain:

$$\begin{array}{rcl}
X_1 & = & \Gamma(\ \bigcup_{\substack{p,q,u,v\in\Sigma^k,\\ r\in P,\ r\to(u,v)}} \{\ ((p,q),c)\ \mid\ w_1,w_2\in\Sigma^*,\ prf_k(w_1)=p, suf_k(w_1)=u, \\
& & \qquad prf_k(w_2)=v,\ suf_k(w_2)=q, \\
& & \qquad c = k_r\cdot\gamma_P^i(M)(w_1)\cdot\gamma_P^i(M)(w_2)\ \}\ ) \\
& = & \Gamma(\ \bigcup_{\substack{p,q,u,v\in\Sigma^k,\\ r\in P,\ r\to(u,v)}} \{\ ((p,q),x)\ \mid\ x = \sum_{\substack{w_1,w_2\in\Sigma^*\text{ such that}\\ prf_k(w_1)=p,\ suf_k(w_1)=u,\\ prf_k(w_2)=v,\ suf_k(w_2)=q}} k_r\cdot\gamma_P^i(M)(w_1)\cdot\gamma_P^i(M)(w_2)\ \}\ ) \\
& = & \Gamma(\ \bigcup_{\substack{p,q,u,v\in\Sigma^k,\\ r\in P,\ r\to(u,v)}} \{\ ((p,q),x)\ \mid\ x = k_r\cdot \sum_{\substack{w_1\in\Sigma^*\text{ such that}\\ prf_k(w_1)=p,\\ suf_k(w_1)=u}} \gamma_P^i(M)(w_1)\ \times \sum_{\substack{w_2\in\Sigma^*\text{ such that}\\ prf_k(w_2)=v,\\ suf_k(w_2)=q}} \gamma_P^i(M)(w_2)\ \}\ ) \\
& = & \Gamma(\ \bigcup_{\substack{p,q,u,v\in\Sigma^k,\\ r\in P,\ r\to(u,v)}} \{\ ((p,q),x)\ \mid\ x = k_r\cdot C(i)((p,u))\cdot C(i)((v,q))\ \}\ ) \\
& = & \delta C_1(i).
\end{array}$$

In a similar manner, we have:

$$\begin{array}{rcl}
X_2 & = & \delta C_2(i), \\
X_3 & = & \delta C_3(i).
\end{array}$$

Therefore, we have:

$$\begin{aligned} R &= \Gamma(\ C(i) \ \cup \ \delta C_1(i) \cup \ \delta C_2(i) \cup \ \delta C_3(i) \ \ ) \\ &= C(i+1), \end{aligned}$$

which completes the proof. □

Let $w = a_1 \cdots a_n$ ($a_i \in \Sigma$, $i = 1, ..., n$) be a string whose concentration at some specified time we want to predict. Using the multisets $C(t)$, we define, for every integer $t \in \mathbf{N}$ and $l_1, l_2 \in \mathbf{N}$ with $0 \leq l_1 < l_2 \leq n$, a real value $A(t, l_1, l_2)$ inductively as follows:

$$\begin{aligned} A(0, l_1, l_2) &= M(a_{l_1+1} \cdots a_{l_2}), \\ A(t+1, l_1, l_2) &= A(t, l_1, l_2) + \delta A_1(t, l_1, l_2) + \delta A_2(t, l_1, l_2) + \delta A_3(t, l_1, l_2), \\ \delta A_1(t, l_1, l_2) &= \sum_{\substack{l_1 < m < l_2, \ \ r \in P, \\ r \rightarrow (suf_k(a_{l_1+1} \cdots a_m), prf_k(a_{m+1} \cdots a_{l_2}))}} k_r \cdot A(t, l_1, m) \cdot A(t, m, l_2), \\ \delta A_2(t, l_1, l_2) &= \sum_{\substack{u, v \in \Sigma^k, \ \ r \in P, \\ r \rightarrow (suf_k(a_{l_1+1} \cdots a_{l_2}), u)}} -k_r \cdot A(t, l_1, l_2) \cdot C(t)((u, v)), \\ \delta A_3(t, l_1, l_2) &= \sum_{\substack{u, v \in \Sigma^k, \ \ r \in P, \\ r \rightarrow (v, prf_k(a_{l_1+1} \cdots a_{l_2}))}} -k_r \cdot C(t)((u, v)) \cdot A(t, l_1, l_2). \end{aligned}$$

We have the following lemma:

Lemma 2 For every $t \geq 0$ and $0 \leq l_1 < l_2 \leq n$, the following equation holds:

$$A(t, l_1, l_2) \ \ = \ \ \gamma_P^t(M)(a_{l_1+1} \cdots a_{l_2}).$$

Proof

We will prove the claim by induction on $t \geq 0$.

In case of $t = 0$, the claim is obtained immediately from the definition. Assume the claim holds for the case of $t \leq i$ and let

$$R \ \ = \ \ \gamma_P^{i+1}(M)(a_{l_1+1} \cdots a_{l_2}).$$

Then, we have:

$$\begin{aligned} R &= \Gamma(\gamma_P^i(M) \cup \delta_P(\gamma_P^i(M)))(a_{l_1+1} \cdots a_{l_2}) \\ &= \gamma_P^i(M)(a_{l_1+1} \cdots a_{l_2}) + \Gamma(\delta_P(\gamma_P^i(M)))(a_{l_1+1} \cdots a_{l_2}) \\ &= A(i, l_1, l_2) + X_1 + X_2 + X_3, \end{aligned}$$

where

$$X_1 = \sum_{\substack{l_1 < m < l_2,\ w_1 = a_{l_1+1}\cdots a_m,\\ w_2 = a_{m+1}\cdots a_{l_2},\ u,v\in\Sigma^k,\\ suf_k(w_1)=u,\ prf_k(w_2)=v,\\ r\in P,\ r\to(u,v)}} k_r\cdot\gamma_P^i(M)(w_1)\cdot\gamma_P^i(M)(w_2),$$

$$X_2 = \sum_{\substack{w_1 = a_{l_1+1}\cdots a_{l_2},\ w_2\in\Sigma^*,\\ u,v\in\Sigma^k,\ prf_k(w_2)=u,\\ suf_k(w_2)=v,\ r\in P,\\ r\to(suf_k(w_1),u)}} -k_r\cdot\gamma_P^i(M)(w_1)\cdot\gamma_P^i(M)(w_2),$$

$$X_3 = \sum_{\substack{w_1\in\Sigma^*,\ w_2 = a_{l_1+1}\cdots a_{l_2},\\ u,v\in\Sigma^k,\ prf_k(w_1)=u,\\ suf_k(w_1)=v,\ r\in P,\\ r\to(v,prf_k(w_2))}} -k_r\cdot\gamma_P^i(M)(w_1)\cdot\gamma_P^i(M)(w_2).$$

Then, we will obtain:

$$\begin{aligned} X_2 &= \sum_{\substack{w_1 = a_{l_1+1}\cdots a_{l_2},\\ u,v\in\Sigma^k,\ r\in P,\\ r\to(suf_k(w_1),u)}} \left(-k_r\cdot\gamma_P^i(M)(w_1) \times \sum_{\substack{w_2\in\Sigma^*,\\ prf_k(w_2)=u,\\ suf_k(w_2)=v}} \gamma_P^i(M)(w_2)\right) \\ &= \sum_{\substack{w_1 = a_{l_1+1}\cdots a_{l_2},\\ u,v\in\Sigma^k,\ r\in P,\\ r\to(suf_k(w_1),u)}} \left(-k_r\cdot A(i,l_1,l_2) \times C(i)((u,v))\right) \\ &= \delta A_2(i,l_1,l_2). \end{aligned}$$

In a similar manner, we have:

$$\begin{aligned} X_1 &= \delta A_1(i,l_1,l_2), \\ X_3 &= \delta A_3(i,l_1,l_2). \end{aligned}$$

Therefore, we have:

$$\begin{aligned} R &= A(i,l_1,l_2) + \delta A_1(i,l_1,l_2) + \delta A_2(i,l_1,l_2) + \delta A_3(i,l_1,l_2) \\ &= A(i+1,l_1,l_2), \end{aligned}$$

which completes the proof. □

By Lemma 1 and Lemma 2, we have the following theorem:

Theorem 2 The CPP problem for simple ligation systems is efficiently computable.
Proof
By Proposition 1, we have for every $t\ge 0$,

$$C(t)\subseteq Prf_k(supp(M))\times Suf_k(supp(M)).$$

Then, it is easy to see that $C(t)$ can be computed in polynomial time with respect to $size(M)$, $size(P)$ and t. Therefore, it is also straightforward to see that for every $0 \leq l_1, l_2 \leq n$, $A(t, l_1, l_2)$ can be computed in polynomial time with respect to $size(M)$, $size(P)$ t and n, where n is the length of the input string w. Thus, the value $\gamma_P^t(M)(w) = A(t, 0, n)$ is efficiently computable. $\square$

Note that in this paper we only deal with real values with finite representations and assume that basic operations of real values, such as addition and multiplication, could be computed in a constant time.

By Theorem 1 and Theorem 2, we have the following main theorem:

Theorem 3 Any decision problem Q which can be computed in polynomial steps using simple ligation systems can be computed in polynomial time by deterministic Turing machines. $\square$

4 Conclusions and Open Problems

In this paper, we proposed a computational mechanism, called *pattern reaction system*, to model chemical reactions of linear molecules, in which every molecule has its concentration. We discuss on the problem of predicting the concentration of a molecule w at the specified time t in a given chemical reaction system and shows its relationship to the computational capability of the system. In particular, we give a polynomial time prediction algorithm for ligation reaction systems, which suggests that the ligation reaction of linear molecules does not have computational capability beyond the class P, even if we consider the concentration of molecules.

One of the problems is that since the proposed model is discretized, there exist numerical errors if we compare it with real chemical reactions systems. Therefore, there still exists a gap between the real system and the proposed one. We think that the model should follow the real kinetics of chemical reactions as far as possible. The authors think that the theory of numerical methods with guaranteed accuracy might give us one of the ways to fill the gap.

Another important research topic is that on the error tolerant molecular computation. Molecular computation is essentially error prone. One of the most basic types of errors might be the errors in the initial concentration of each molecule and in the condition parameters (e.g., temperature) of chemical reactions. The theory and methods for the concentration prediction problem might give an analytical method for making an error tolerant molecular computer, since they give the relationship between the input parameters and the concentration of final products.

The current paper discusses only on a simple version of the ligation reaction. It is an interesting open research topic to generalize the method presented in this paper and investigate and characterize a class of chemical reactions whose CPP is efficiently solvable.

References

[Adl94] Leonard M. Adleman, Molecular computation of solutions to combinatorial problems, *Science*, 266:1021–1024 (1994)

[Adl96] Leonard M. Adleman, On Constructing A Molecular Computer, in *DNA Based Computers, Proc. of a DIMACS Workshop*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, R. J. Lipton and E. B. Baum (Eds.), pp. 1-21 (1996)

[ASY92] S. Arikawa, T. Shinohara and A. Yamamoto. Learning Elementary Formal Systems. *Theoretical Computer Science*, **95**, pp.97-113, 1992

[BB92] Gérard Berry and Gérard Boudol, The chemical abstract machine. *Theoretical Computer Science*, Vol.96, No.1, pp. 217-248, 1992.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company (1979)

[HN99] Masami Hagiya and Akio Nishikawa, Concurrency Calculi from the Viewpoint of Molecular Computing – Making Chemical Abstract Machines More Chemical –, Journal of Japan Society for Fuzzy Theory and Systems, Vol.11, No.1, pp.2-13, 1999 (in Japanese).

[Hea87] Tom Head, Formal language theory and DNA : An analysis of the generative capacity of specific recombinant behaviors, *Bulletin of Mathematical Biology*, 49:737–759 (1987)

[KYSM97] Satoshi Kobayashi, Takashi Yokomori, Gen-ichi Sampei and Kiyoshi Mizobuchi, DNA Implementation of Simple Horn Clause Computation, in *Proc. of IEEE International Conference on Evolutionary Computation*, pp.213-217 (1997)

[Kob99] Satoshi Kobayashi, Horn Clause Computation with DNA Molecules, *Journal of Combinatorial Optimization*, Vol.3, pp.277-299, 1999. in *Proc. of IEEE International Conference on Evolutionary Computation*, pp.213-217 (1997)

[KR99] S. N. Krishna, R. Rama, On the power of P systems with sequential and parallel rewriting, manuscript, 1999.

[Mih97] Valeria Mihalache, Prolog Approach to DNA Computing, in *Proc. of IEEE International Conference on Evolutionary Computation*, pp.249-254 (1997)

[OR98] Mitsunori Ogihara and Animesh Ray, Minimum DNA Computation Model and Its Computational Power, in *Proc. of 1st Workshop on Unconventional Models of Computation*, pp.309-322 (1998)

[Pau95] Gh. Păun, Regular extended H systems are computationally universal, *J. Inform. Process. Cybern., EIK,*, (1995) [Pau98] Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, in press, and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).

[Pau00] Gheorghe Păun, Computing with membranes (P Systems): Twenty Six Research Topics. manuscript, 2000.

[PP00] A. Păun, M. Păun, On the membrane computing based on splicing, submitted, 2000.

[PRS99] G. Păun, G. Rozenberg, A. Salomaa, DNA Computing – New Computing Paradigms, Springer-Verlag, 1998.

[PY99] Gh. Păun, T. Yokomori, Membrane computing based on splicing, *Preliminary Proc. of Fifth Intern. Meeting on DNA Based Computers* (E. Winfree, D. Gifford, eds.), MIT, June 1999, 213–227.

[Rei95] John H. Reif, Parallel Molecular Computation: Models and Simulations, in *Proc. of Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA95)*, ACM, Santa Barbara, 213-223 (1995) Also to appear in Algorithmica, special issue on Computational Biology, 1998.

[Rot96] Paul Wilhelm Karl Rothemund, A DNA and restriction encyme implementation of Turing Machine, in *DNA Based Computers, Proc. of a DIMACS Workshop*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, R. J. Lipton and E. B. Baum (Eds.), pp. 75-119 (1996)

[Shi94] T. Shinohara. Rich Classes Inferable from Positive Data : Length Bounded Elementary Formal Systems. *Information and Computation*, **108**, pp.175-186, 1994

[Smu61] Raymond M. Smullyan, Theory of Formal Systems, Annals of Mathematics Studies, **47**, revised edition, Princeton University Press, 1961.

[Win96] Eric Winfree, Complexity of Restricted and Unrestricted Models of Molecular Computation, in *DNA Based Computers, Proc. of a DIMACS Workshop*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, R. J. Lipton and E. B. Baum (Eds.), pp. 187-198 (1996)

[Zan00a] Cl. Zandron, Two normal forms for rewriting P systems, manuscript, 2000.

[Zan00b] Cl. Zandron, Priorities and variable thickness of membranes in rewriting P systems, manuscript, 2000.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 124 - 137.

Computing with Simple P Systems

S.N. Krishna
Department of Mathematics
Indian Institute of Technology, Madras
Chennai-600036,Tamil Nadu, India
E-mail : maph9801@violet.iitm.ernet.in

Abstract

The P Systems have been recently introduced as a new model for distributed parallel computing. We describe in this paper, a new variant of P Systems: Simple P Systems. We consider two variants of Simple P systems: Rewriting simple P systems and splicing simple P systems. Both the variants are proved to be computationally complete. In the case of rewriting simple P systems, computational completeness is achieved using two membranes with priorities, whereas in splicing simple P systems, the same is achieved by systems of degree seven and no priorities.

Keywords: Membrane structure, Recursively enumerable set, Simple P system, Matrix grammar, Splicing, Natural Computing

1 Introduction

In this paper, we consider a new model of computation, called *P Systems* or *Super Cell Systems*. In this model, a computation is performed by computing cells. Membranes are used to enclose computing cells in order to make them independent computing units. Also, a membrane serves as a communication channel between a given cell and other cells adjacent to it. The name "membrane" is suitable here because also biological membranes surrounding biological cells have these two functions. The structure of cells is recursive; computing cells may contain other computing cells. In this way, through the inclusion relation, a hierarchical structure is imposed for the whole computing unit. If a cell does not contain other cells, it is called *elementary*. The membrane surrounding the cell which is the highest in the hierarchy is called the *skin*. The structure of cells is dynamic: the cells may be "removed" - this is achieved by dissolving the membrane surrounding a cell to be removed. A single cell is a complete computing unit in the sense that it has its own computing program. To be more precise, this computing program governs the area of a given cell included between the membrane of a cell and the membranes of the cells included in the given cell - this area is referred to as a *region*.
A *membrane structure* is a construct consisting of several membranes placed in a

unique skin membrane; we formalize a membrane structure by means of well-formed paranthesized expressions, strings of correctly matching parantheses, placed in a unique pair of matching parantheses;each pair of matching parantheses correspond to a membrane. This notion is similar to that used by the chemical abstract machine, [1]. The membranes are labeled in a one-to-one manner. Each membrane identifies a region, delimited by it and the membranes inside it(if any). If in the regions delimited by the membranes we place multisets of objects from a specified finite set V, then we obtain a *super cell.* (A *multiset* over V is a mapping $M : V \to N$; N is the set of natural numbers. M(a), for a in V is the multiplicity of a in the multiset M).

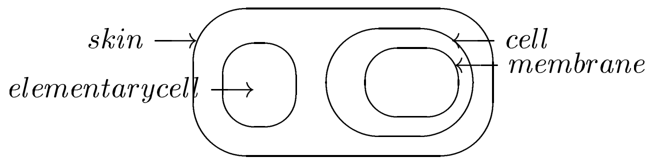


Figure 1: A membrane structure.

A membrane structure can also be represented by means of a Venn diagram as above. The above figure corresponds to the membrane structure [[ ] [[ ]]]. If we have a membrane structure $[_1[_2]_2[_3[_4[_5]_5]_4]_3]_1$, we say membranes 2, 3, 4, 5 are inside 1; membrane 4 is immediately inside 3, membrane 5 is inside 3 and so on. More formally, a P system or Super cell system of degree $m, m \geq 1$, is a construct

$$\Pi = (V, T, C, \mu, M_1, M_2, \ldots, M_m, (R_1, \rho_1), (R_2, \rho_2), \ldots, (R_m, \rho_m))$$

where:

(1) V is the total alphabet of the system; its elements are called *objects*;

(2) T $\subseteq$ V (the output alphabet or terminal alphabet);

(3) C $\subseteq$ V, C $\cap$ T = ϕ (catalysts);

(4) μ is a membrane structure consisting of m membranes, with the membranes and the regions labeled in a one-to-one manner with elements in given set; here we always use labels 1,2,...,m;

(5) $M_i, 1 \leq i \leq n$, are multisets over V associated with the regions $1, 2, \ldots, m$ of μ;

(5) $R_i, 1 \leq i \leq m$ are finite sets of *evolution rules* over V associated with the regions $1, 2, \ldots, m$ of $\mu; \rho_i$ is a partial order relation over R_i, specifying a priority relation among rules of R_i. An evolution rule is a pair (u, v) which we usually write in the form $u \longrightarrow v$, where u is a string over V and $v = v'$ or $v = v'\delta$, where v' is a string over

$$(V \times \{here, out\}) \cup (V \times \{in_j \mid 1 \leq j \leq m\}),$$

and δ is a special symbol not in V. The length of u is called the radius of the rule $u \longrightarrow v$. (The strings u, v are understood as representations of multisets over V).

If Π contains rules of radius greater than one, then we say that Π is a system with cooperation. Otherwise, it is a non-cooperative system. A particular class of cooperative systems is that of catalytic systems; the only rules of radius greater than one are of the form $ca \longrightarrow cv$, where $c \in C, a \in V - C$, and v contains no catalyst; moreover, no other evolution rules contain catalysts (there is no rule of the form $c \longrightarrow v$ or $a \longrightarrow v_1cv_2$, for $c \in C$). The membrane structure and the multisets in Π constitute the *initial configuration* of the system. We can pass from one configuration to another one by using the evolution rules. This is done in parallel; all objects, from all membranes, which can be the subject of local evolution rules, as prescribed by the priority relation should evolve simultaneously. The priority checking is done as follows : we take a rule for which there is no rule of a higher priority and assign to it the objects to which it can be applied; we repeat this operation with the rule of a maximal priority which can be applied to the objects which were not assigned yet to rules (the objects are assigned only once to a rule). We continue till no further rule $u \longrightarrow v$ exists such that u is included in the multiset of non-assigned objects. All objects which were assigned to rules will evolve by using these rules, in one step all.
The use of a rule $u \longrightarrow v$ in a region with a multiset M means to subtract the multiset identified by u from M, providing that the multiset identified by u is included in M, then to follow the prescriptions of v: If an object appears in v as $(a, here)$, then it remains in the same region; if we have (a, out), then a copy of the object a will be introduced in the membrane placed immediately outside the region of the rule $u \longrightarrow v$; if we have $(a, in\ i)$, then a copy of a is introduced in the membrane with the label i, providing that it is adjacent to the region of the rule $u \longrightarrow v$, otherwise the rule cannot be applied; if the special symbol δ appears in v, then the membrane which delimits the region where we work is dissolved; in this way, all the objects in this region become elements of the region placed immediately outside, while the rules of the dissolved membrane are removed. The rules are applied in parallel, an object introduced by a rule cannot evolve at the same step by means of another rule. Note that the catalysts cannot pass from a region to another one by indications of the form (c, out) or (c, in_j), but only by membrane dissolving actions.
A sequence of transitions between configurations of a given P System Π is called a *computation* with respect to Π. A computation is successful iff it halts, that is there is no rule applicable to the objects present in the last configuration of the computation. The result of a successful computation is assigned as follows: we observe the system from outside and collect the objects ejected from the skin membrane, in the order they are ejected. Using these objects, we form a string. When several objects are ejected at the same time, then any permutation of them is considered. The result of a successful computation can also be considered as $\psi_T(w)$, where w describes the multiset of objects from T sent out of the system. The set of vectors

$\psi_T(w)$ for w describing the multiset from T sent out of the system at the end of a halting configuration is denoted $Ps(\Pi)$ and we say that it is *generated* by Π. (If $V = \{a_1, a_2, \ldots, a_m\}$, then the *Parikh mapping* associated with V is $\psi_V : V^* \longrightarrow N^n$ defined by $\psi_V(x) = (\mid x \mid_{a_1}, \mid x \mid_{a_2}, \ldots, \mid x \mid_{a_m})$ for $x \in V^*$. $\psi_L(V)$ is called the *Parikh set* of $L \subseteq V^*$. The family of Parikh sets of languages in a family F is denoted by Ps F). Similarly, the family of length sets of languages in a family F is denoted by Ls F; and the the permutation closure of a language L is denoted $p(L)$.(For a set $M \subseteq N^k$, consider the language $l(M) \subseteq V^*$, for $V = \{a_1, a_2, \ldots, a_k\}$, defined by $l(M) = \{w \in V^* \mid \psi_V(w) \in M\}$. Then $p(L)$ is used to denote the language $l(\psi_V(L)))$. There is yet another way to assign the result of a successful computation: designate some membrane as the output membrane, and this membrane should be an elementary one in the last configuration. (Note that the output membrane was not necessarily an elementary one in the initial configuration). In this case, the total number of objects present in the output membrane of the halting configuration or $\psi_T(w)$ where w represents the multiset of objects from T present in the output membrane in a halting configuration is the resultant of a successful computation.
In the following sections, we consider two variants of Simple P systems; these variants differ from one another in the way of application of rules. In rewriting and splicing simple P systems, the objects considered are strings. The evolution rules used in rewriting simple P systems are rewriting rules and those used in splicing simple P systems are splicing rules. Many variants of P systems are considered and investigated in [2-7], [9-12]. All of them have been proved to be computationally universal. Some variants [3], [7] are also capable of solving hard problems.

2 Simple P Systems

In this section, we define a new variant of P systems : Simple or Uniform P Systems. The idea of having this system and to study its properties was suggested as an open problem in [8]. These are systems for which we have a single set of rules for all the membranes. Unlike usual P systems for which we have local evolution rules for each of the membranes, in Simple P systems we have a set of "global" rules, in the sense that it is applicable to all membranes. In the earlier systems, the dissolvation of a particular membrane resulted in the loss of the corresponding set of rules; whereas in Simple P systems, the rules are never lost. That is, the rules are pertaining to the objects alone; the earlier systems had rules pertaining to the membranes and the objects within each membrane. Formally, we define a Simple P System as follows:

Definition 2.1 *A Simple P System of degree n, n≥1, is a construct*

$$\Pi = (V, T, C, \mu, w_1, w_2, \ldots, w_n, (R, \rho))$$

where:

(1) V is the total alphabet of the system; its elements are called objects;

(2) $T \subseteq V$ (the output alphabet or terminal alphabet);

(3) $C \subseteq V$, $C \cap T = \phi$ *(catalysts);*

(4) $w_i, 1 \leq i \leq n$, *are multisets over* V *associated with the regions* $1, 2, \ldots, n$ *of* μ*;*

(5) R *is the set of evolution rules over* V *associated with all the regions of* μ; ρ *is a partial order relation over R; specifying a priority relation among rules of R. An evolution rule is a pair* (u, v) *which we usually write in the form* $u \longrightarrow v$, *where* u *is a string over* V *and* $v = v'$ *or* $v = v'\delta$, *where* v' *is a string over*

$$(V \times \{here, out\}) \cup (V \times \{in_j \mid 1 \leq j \leq m\}),$$

and δ *is a special symbol not in* V*. The length of* u *is called the radius of the rule* $u \longrightarrow v$*.*

Note that we refer to a system with just one set of rules and objects of any kind as a simple P system. If the objects are "atomic" in nature, that is if we consider multisets of objects, and if there is only one set of rules, we call it a transition simple P system. Since we have just one set of rules, the following points must be noted: if there is a rule $a \longrightarrow (v, in\ j)$, this is applicable only in the membrane surrounding j; similarly, a rule involving δ is applicable in all membranes other than the skin membrane. Rules with target "here" and "out" are globally applicable: that is to all membranes. The language generated is defined similarly as above. That is, we collect all the objects over T coming out of the system at the end of a halting configuration.

3 Examples

In this section, we give some examples of Transition Simple P systems.

Example 3.1 *First we give an example to show how transitions take place in a Simple P System. Consider the system* $\Pi = (\{A, B, E, a, d, f\}, \{a, f\}, \{c\}, [_1[_2]_2[_3]_3]_1, \{cA\}, \{Bd\}, \{E\}, (R, \rho))$ *where the rules and the priorities are as follows:*
$r_1 : cA \longrightarrow c(a, out)$; $r_2 : B \longrightarrow B(AA, out)$; $r_3 : B \longrightarrow B$; $r_4 : B \longrightarrow \lambda$; $r_5 : d \longrightarrow d$; $r_6 : d \longrightarrow f\delta$; $r_7 : E \longrightarrow Ef$; $r_8 : E \longrightarrow f\delta$; $r_9 : f \longrightarrow (f, out)$. *The priorities are* $r_1 > r_2$. *We start working by applying rules* r_1, r_3 *or* r_4, r_5 *or* r_6, r_7 *or* r_8. *Suppose* r_1, r_3, r_5, r_7 *are applied.* r_2 *can be applied only when* r_1 *cannot be applied; that is when there is no copy of A in the skin membrane. The system can come to a halt only after applying* r_6 *and* r_8. *If* r_2 *is applied after* r_6, r_1 *is no longer applicable, as the A's will go out of the system. The following steps will clarify the way transitions take place.*
$[_1cA[_2Bd]_2[_3E]_3]_1 \Longrightarrow a[_1c[_2Bd]_2[_3Ef]_3]_1 \Longrightarrow a[_1fcAA[_2Bd]_2[_3Ef]_3]_1 \Longrightarrow$
$a(af\ or\ fa)[_1fcA[_2Bd]_2[_3Ef]_3]_1 \Longrightarrow a(af\ or\ fa)(af\ or\ fa)[_1fc[_2Bd]_2[_3Ef]_3]_1 \Longrightarrow$
$a(af or fa)(af or fa)f[_1c[_2Bd]_2ff]_1 \Longrightarrow a(af or fa)(af or fa)fff[_1cBf]_1 \Longrightarrow$
$a(af or fa)(af or fa)ffff[_1c\lambda]_1$. *The rules applied here are in the following order: Step* 0*:Initial configuration Step* $1 : r_1, r_3, r_5, r_7$ *Step* $2 : r_2, r_5, r_7, r_9$ *Step* 3, $4 : r_1, r_3, r_5, r_7, r_9$ *Step* $5 : r_3, r_5, r_8, r_9$ *Step* $6 : r_3, r_6, r_9$ *Step* $7 : r_4, r_9$. *The objects a and f collected outside at the end of Step 7 are the resultant of this computation.*

After Step 7, the system halts as no more rule is applicable. Note that the system can be made to halt in any step after applying rules r_6 and r_8.

Example 3.2 *Consider the Simple P System* $\Pi = (\{A, B, C, a, b, c\}, \{a, b, c\}, \phi, [_1[_2]_2[_3]_3]_1, \{A\}, \{B\}, \{C\}, (R, \rho))$ *with no catalysts and having priorities. The rules are:* $A \longrightarrow aA, B \longrightarrow bB, C \longrightarrow cC, A \longrightarrow a, B \longrightarrow b, C \longrightarrow c, a \longrightarrow a(out)$, $b \longrightarrow b(out), c \longrightarrow c(out)$. *The priorities for the rules are* $A \longrightarrow a > B \longrightarrow bB, C \longrightarrow cC$; $B \longrightarrow b > A \longrightarrow aA, C \longrightarrow cC$; $C \longrightarrow c > A \longrightarrow aA, B \longrightarrow bB$. *Clearly, the language generated is* $L(\Pi) = \{x \in \{a, b, c\}^* \mid \ \mid x \mid_a = \mid x \mid_b = \mid x \mid_c\}$. *The priorities ensure that the evolutions corresponding to* A, B, C *terminate at the same time. The terminals* a, b, c *leave the system using the rules* $a \longrightarrow a(out), b \longrightarrow b(out), c \longrightarrow c(out)$.

Example 3.3 *Consider the following system of degree two, with priorities and no cooperation* $\Pi = (\{A, A', B, a, b, c, d\}, \{a, b, c, d\}, [_1[_2]_2]_1, \{A\}, \phi, (R, \phi))$ *where R consists of the following rules:*
$r_1 : A \longrightarrow A(a, out)(B, in\ 2)$; $r_2 : A \longrightarrow B(b, out)(A', in\ 2)$; $r_3 : c \longrightarrow (c, out)$; $r_4 : d \longrightarrow (d, out)$; $r_5 : B \longrightarrow Bc$; $r_6 : A' \longrightarrow d\delta$; $r_7 : B \longrightarrow \lambda$; $r_8 : c \longrightarrow c$. *The priorities are :* $r_1, r_2 > r_3, r_7$; $r_6 > r_7$; $r_3, r_4 > r_5$. *We start working in the skin membrane, where there is available a copy of A. By using the rule* $A \longrightarrow A(a, out)(B, in\ 2)$, *we reproduce the object A in membrane one and send out a copy of a, and we introduce a copy of B in membrane two. From now on, both in the inner and outer membranes, we have applicable rules. At each step in membrane one, we repeat the the previous operation, while in the inner membrane we produce a copy of c from each available copy of B in parallel.(The rule* r_7 *is not applicable because of the priority). For instance, after five steps, we have five copies of a outside, one copy of A in membrane one, five copies of B in membrane two, and 4+3+2+1=10 copies of c in membrane two. In any moment, the rule* $A \longrightarrow B(b, out)(A', in\ 2)$ *can be applied. One copy of B is kept in membrane one, a copy of b is sent outside (hence the string collected becomes* $a^n b$ *for some* $n \geq 0$.*) and a copy of* A' *is sent to membrane two. At the same time with the use of the rule* $B \longrightarrow Bc$ *for all copies of B present here, we have to apply the rule* $A' \longrightarrow d\delta$. *Membrane two is dissolved, its contents are left free in membrane one, where the rules* $c \longrightarrow (c, out)$, $d \longrightarrow (d, out)$ *and* $B \longrightarrow \lambda$ *are applicable. Since the d and the c's are sent out in parallel, outside the system we get* $n(n+1)/2$ *copies of c, one copy of d. Consequently, as an output we can consider any of the strings* $a^n bc^i dc^j$ *for* $n \geq 0$ *and* $i + j = n(n+1)/2$. *That is, the language obtained in this way is* $L(\Pi) = \{a^n bc^i dc^j \mid n \geq 0,\ and\ i + j = n(n+1)/2, i, j \geq 0\}$.

4 Simple P Systems based on Rewriting

In this section, we consider Simple P systems in which the objects are described by finite strings over a finite alphabet. The evolution of an object will then correspond to a transformation of the string. In this section, we consider transformations in the form of rewriting steps, as usual in formal language theory. Consequently, the evolution rules are given by rewriting rules. Assume that we are given an alphabet V. As in the previous section, here also the rules are provided with indications on the target membrane. Always we use only context-free rules. Thus rules of the form $X \longrightarrow v(tar)$ where $tar \in \{here, out, in\ j\}$ are used with the obvious meaning: the string produced by using this rule will go to the membrane indicated by tar. A string is now an unique object, hence it passes through membranes as a unique entity, its symbols do not follow different itineraries as it was possible for the objects in a multiset; of course, in the same region, we have several strings at the same time. In this way, we obtain a language generating mechanism of the form

$$\Pi = (V, T, \mu, w_1, w_2, \ldots, w_m, (R, \rho))$$

where V is the total alphabet, T is the terminal alphabet or output alphabet, μ is the membrane structure, $w_1, w_2, \ldots, w_m$ are finite languages over V present in membranes $1, 2, \ldots, m$, and R is a finite set of context-free rules of the form $X \longrightarrow v(tar)$, with $X \in V, v \in V^*, tar \in \{here, out, in\ j\}$ and ρ is a partial order relation over R. We call such a system a *rewriting simple P system.* The language generated by Π is denoted by $L(\Pi)$ and consists of all strings over T^* sent out of the system at the end of a halting configuration. A computation is defined similarly as in the previous section, with the differences specific to an evolution based on rewriting : we start from the initial configuration of the system and proceed iteratively, by transition steps done by using the rules in parallel, to all strings which can be rewritten obeying the priority relations, and collecting the strings sent out of the system. Note that each string is processed by one rule only, the parallelism refers to processing simultaneously all available strings by all applicable rules. If several rules can be applied to a string, at several places each, then we take only one rule and only one possibility to apply it and consider the obtained string as the next state of the object described by the string. The evolution of strings are not independent of each other, but interrelated in two ways: if we have a priority $r_1 > r_2$, and if r_1 is applicable to a string x, the application of r_2 to another string y present in the system is forbidden; even without priorities, if a string x can be rewritten forever, then the system never halts and all strings are lost, irrespective of all the strings sent out. If non-context free rules or rules of radius greater than one are applied, then the system is said to be cooperative. As in the previous section, a rule with target $in\ j$ is applicable only in a membrane adjacent to j. Here we do not introduce the membrane dissolving action as it is not required for computational completeness. We denote by $ERSP_m(\alpha, \beta)$, the language generated by rewriting Simple P systems of degree atmost m, $\alpha \in$ {Pri, n Pri}, $\beta \in$ {Coo, n Coo}, where "Coo" stands for cooperative or non context-free rules, "n Coo" stands for non cooperative or context-free rules. The union of all families $ERSP_m(\alpha, \beta)$ is denoted by $ERSP(\alpha, \beta), \alpha \in \{Pri, nPri\}, \beta \in \{Coo, nCoo\}$.

Theorem 4.1 *$CF = ERSP_1$(n Pri, n Coo), and $CF \subset ERSP_2$(n Pri, n Coo).*

Proof : The equality can be proved in a similar manner as in [5]. To prove the strict inclusion, consider the rewriting simple P system $\Pi = (\{A, B, C, a, b, c\}, \{a, b, c\}, [_1[_2]_2]_1, AB, \phi, (R, \phi))$ where the rules are $A \longrightarrow (aAb, out), B \longrightarrow (cB, in\ 2), A \longrightarrow (ab, out)$, $B \longrightarrow c$. Clearly, the language generated is $L(\Pi) = \{a^n b^n c^n \mid n \geq 1\}$.

Theorem 4.2 *$RE \subseteq ERSP_2$(Pri, n Coo).*

Proof : Let G = (N, T, S, M, F) be a matrix grammar in binary normal form. Let there be k matrices numbered $m_1, m_2, \ldots, m_k$. We construct the rewriting simple P system $\Pi = (V, T, \mu, w_1, w_2, (R, \rho))$ where $V = N_1 \cup N_2 \cup \{Y_i, Z_{Y_i}, Y_i', Z_{Y_i'}, i, i', i'', i''', A_i, A_i', A_i'', D_{A_i} \mid Y \in N_1, A \in N_2, 1 \leq i \leq k\}$, $\mu = [_1[_2]_2]_1, w_1 = XA$ such that $(S \longrightarrow XA)$ is a matrix of type 1 in G, $w_2 = \lambda$. The rules are as follows:
$r_1 : \{X \longrightarrow Y_i \mid m_i : (X \longrightarrow Y, A \longrightarrow x)$ is a matrix of Type 2 in G$\}$
$r_2 : \{X \longrightarrow Y_i' \mid m_i : (X \longrightarrow Y, A \longrightarrow \dagger)$ is a matrix of Type 3 in G$\}$
$r_3 : \{X \longrightarrow i' \mid m_i : (X \longrightarrow \lambda, A \longrightarrow x)$ is a matrix of Type 4 in G$\}$
$r_4 : \{Y_i \longrightarrow i''' Z_{Y_i} \mid Y \in N_1, 1 \leq i \leq k\}$; $r_5 : \{i' \longrightarrow i'' \lambda_i \mid 1 \leq i \leq k\}$
$r_6 : \{A \longrightarrow (A_i, in\ 2) \mid m_i : (X \longrightarrow Y/\lambda, A \longrightarrow x)$ is a matrix of type 2 or 4 in G$\}$
$r_7 : \{A \longrightarrow A_i' D_{A_{i+1}} \ldots D_{A_k} \mid m_i, m_{i+1}, \ldots, m_k$ are type 3 matrices having a rule for $A \in N_2\}$
$r_8 : \{Y_i' \longrightarrow (Z_{Y_i'}, in\ 2)\}$; $r_9 : \{A' \longrightarrow A \mid A \in N_2\}$
$r_{10} : \{A_i \longrightarrow i A_i'' \mid A \in N_2, 1 \leq i \leq k\}$; $r_{11} : \{a \longrightarrow (a, out) \mid a \in T\}$
$r_{12} : \{Z_{Y_i} \longrightarrow Y\} \cup \{\lambda_i \longrightarrow \lambda\}$; $r_{13} : \{i \longrightarrow \lambda \mid 1 \leq i \leq k\}$; $r_{14} : \{Y_i' \longrightarrow Y_i' \mid Y \in N_1\}$
$r_\dagger : \{\dagger \longrightarrow \dagger\}$; $r_{i_1} : \{Z_{Y_i'} \longrightarrow (Y, out) \mid 1 \leq i \leq k\}$
$r_{i_2} : \{A_i' \longrightarrow (\dagger, out) \mid A \in N_2, 1 \leq i \leq k\}$; $r_{i_3} : \{A_i'' \longrightarrow (x, out) \mid 1 \leq i \leq k\}$
$r_{i_4} : \{i'' \longrightarrow i \mid 1 \leq i \leq k\} \cup \{i''' \longrightarrow i \mid 1 \leq i \leq k\}$
$r_{j_1}' : \{A_j' \longrightarrow A', D_{A_j} \longrightarrow \lambda \mid 1 \leq j \leq k\}$; $r_{j_2}' : \{j \longrightarrow \dagger \mid 1 \leq j \leq k\}$
The priorities for the rules are as follows:
$\{r_1, r_2, r_3, r_4, r_5, r_9, r_{10}, r_{i_1}, r_{j_1}', r_{i_3} > r_6, r_7$; $r_{13} > r_1, r_2, r_3$; $r_{i_4} > r_{j_2}', r_{10}$; $r_{i_3}, r_{10}, r_{12} > r_{13}$; $r_{12} > r_{i_3}, r_7$; $r_6 > r_{i_4}, r_{12}$; $r_{10}, r_{j_2}' > r_{i_3}, i \neq j$; $r_{10} > r_{j_2}'$; $r_7 > r_8, r_{j_1}'$; $r_{14} > r_6$; $r_9 > r_{i_1}$; $r_{j_1}' > r_9, r_{i_1}, i \neq j$; $r_{i_2} > r_{i_1}$; $r_8 > r_{j_1}', r_{i_2}\}$
The system works as follows: Suppose at some instant, we have a string $Xw, X \in N_1, w \in (N_2 \cup T)^*$ in membrane one. One of the rules r_1, r_2, r_3 can be applied to X. The rules r_1 or r_3 mean that we are simulating a matrix of type 2 or 4. First we consider simulating a type 2 matrix. In this case we apply r_1 to X. In the next two steps, we apply r_4 and r_6 and the string moves to membrane two.(note that if r_4 is applied, the symbol Z_{Y_i} is introduced and this prevents the application of r_7. so if a rule corresponding to a type 2 matrix is applied to X, then to symbols of N_2 also, rules corresponding to type 2 matrices are applied). Now the rule r_{i_4} is applied changing i''' to i. Then the rules $r_{10}, r_{j_2}', r_{12}, r_{i_3}, r_{13}$ are applied in order (due to the priorities) and the string reaches membrane one if the symbol A for which the rule r_6 was applied corresponds to X. Otherwise, the rule r_{j_2}' is applied and the computation never halts. Thus, if the simulation is correctly done, the string reaches membrane one after successfully simulating a type 2 matrix.
Now, we consider simulating a type 3 matrix. In this case, r_2 is applied to X. In the next step, we apply r_7 (here r_6 cannot be applied as $r_{14} > r_6$). By this rule, we simulate all symbols of N_2 corresponding to type 3 matrices, and all matrices of type 3 corresponding to each symbol. Once this is done, the string moves to membrane two using r_8 ($r_8 > r_{j_1}', r_{i_2}$). Here, we apply r_{j_1}' ($r_{j_1}' > r_{i_1}$) to check if the symbol $A \in N_2$ corresponding to X occurs or not. The A_j''s converted to A''s are further changed to A in the next step using r_9. After this step, if any more A_i''s remain (which mean that the A corresponding to X occurs), then r_{i_2} is applied and the

computation never halts. Otherwise, the string goes to the skin membrane using r_{i_1}, replacing $Z_{Y_i'}$ by Y. In this way, an appearance checking rule is also correctly simulated. The simulation of a type 4 matrix is similar to that of type 2. The string can leave the system using r_{11}. If the string which comes out is purely over terminals, it is listed in the language. Hence, $RE \subseteq ERSP_2(Pri, nCoo)$.

Theorem 4.3 $RE \subseteq ERSP(nPri, nCoo)$

Proof : Let G = (N, T, S, M, F) be a matrix grammar in binary normal form. Let $m_1, m_2, \dots, m_k$ be matrices of type 2 or 4 and $m_{k+1}, \dots, m_l$ be matrices of type 3. We construct the rewriting simple P system $\Pi = (V, T, \mu, w_0, w_1, w_{1'}, \dots, w_l, w_{l'}, (R, \phi))$ where $V = N_1 \cup N_2 \cup \{d, d', d'', \dagger\} \cup \{A_i, A_j', Y' \mid A \in N_2, Y \in N_1,\ 1 \leq i \leq k,\ k+1 \leq j \leq l\}$, $\mu = [_0[_1[_{1'}]_{1'}]_1 \dots [_l[_{l'}]_{l'}]_l]_0$, $w_0 = XA$ such that $(S \longrightarrow XA)$ is a matrix of type 1 in G, $w_i = \lambda$ for all other i. The rules are as follows:
$\{X \longrightarrow (Y, in\ i) \mid m_i$ is a type 2 matrix having the rule $X \longrightarrow Y$ for $X \in N_1, 1 \leq i \leq k\}$;
$\{X \longrightarrow (\lambda, in\ i) \mid m_i$ is of type 4 having the rule $X \longrightarrow \lambda$ for $X \in N_1, 1 \leq i \leq k\}$;
$\{X \longrightarrow (Y', in\ i) \mid m_i$ is of type 3 having the rule $X \longrightarrow Y$ for $X \in N_1, k+1 \leq i \leq l\}$;
$\{A \longrightarrow (A_i, in\ i') \mid A \in N_2, 1 \leq i \leq k$,and m_i is of type 2 or 4 having a rule for $A\}$;
$r_{A_i} : \{A_i \longrightarrow d'(dx, out) \mid m_i$ is a matrix having the rule $A \longrightarrow x, A \in N_2, 1 \leq i \leq k\}$;
$r_{d''} : \{d'' \longrightarrow \dagger\}$; if r_{A_i} is applicable
$r'_{d''} : \{d'' \longrightarrow \lambda\}$; if $r_{d''}$ is not applicable
$\{A \longrightarrow (A_i', in\ i') \mid m_i$ has a rule for $A \in N_2, k+1 \leq i \leq l\}$;
$\{A_i' \longrightarrow \dagger \mid k+1 \leq i \leq l\}$; $\{d \longrightarrow (\lambda, out)\}$; $\{d' \longrightarrow d''\}$;
$\{Y' \longrightarrow Y(Y, out) \mid Y \in N_1\}$; $\{a \longrightarrow (a, out) \mid a \in T\}$; $\{\dagger \longrightarrow \dagger\}$;
The system works as follows : Suppose that at some instant we have a string $Xw, X \in N_1, w \in (N_2 \cup T)^*$ in the skin membrane. Then, we can apply one of the rules $X \longrightarrow (Y, in\ i), X \longrightarrow (Y', in\ i)$ or $X \longrightarrow (\lambda, in\ i)$. If the first rule is applied, it means we are simulating a type 2 matrix. In this case, the string moves to membrane $i, 1 \leq i \leq k$. Now the rule $A \longrightarrow (A_i, in\ i')$ can be applied to some $A \in N_2$ provided it corresponds to matrix m_i. In the next step, we apply r_{A_i} which leaves a copy of the string with A_i replaced by d' in membrane $i', 1 \leq i \leq k$ and another copy of the string comes out to membrane i with A_i replaced by dx, where d is a new symbol and x corresponds to the rule $A \longrightarrow x$ in m_i. In the next step, the copy of the string in membrane i can either go to the skin membrane using $d \longrightarrow (\lambda, out)$ or again move to membrane i' using the rule $A \longrightarrow (A_i, in\ i')$. In the former case, the simulation is correct and the rules $d' \longrightarrow d'', r'_{d''}$ can be applied in consequent steps. The copy of the string remaining in membrane i' will be inactive during the rest of the computation. If on the other hand, the rule $A \longrightarrow (A_i, in\ i')$ is applied to the string in membrane i instead of $d \longrightarrow (\lambda, out)$, then along with it we apply $d' \longrightarrow d''$ to the string in membrane i'. The symbol d'' then takes care of this wrong simulation; in the next step, the rule $r_{d''}$ is applied and the computation never stops.
Now we will see how a type 3 matrix is simulated. In this case, the rule $X \longrightarrow (Y', in\ i), k+1 \leq i \leq l$ is applied and the string moves to membrane $i, k+1 \leq i \leq l$. Now the applicable rules are $A \longrightarrow (A_i', in\ i')$ or $Y' \longrightarrow Y(Y, out)$. If the second

rule is applied, a copy of the string with Y' replaced with Y is placed in membrane i itself, while another copy of the same string is sent to the skin membrane. To the copy of the string in membrane i, the rule $A \longrightarrow (A_i', in\ i')$ can be applied(provided there exists an $A \in N_2$ in the string which has a rule in $m_i, k+1 \leq i \leq l$). If there is no such A in the string, the copy of the string in membrane $i, k+1 \leq i \leq l$ remains as such; the computations can be continued with the other copy which has been sent to the skin membrane. If on the other hand, such an A exists, the string goes to membrane $i', k+1 \leq i \leq l$, with A replaced by A_i'. In the next two steps, the rules $A_i' \longrightarrow \dagger$ and $\dagger \longrightarrow \dagger$ are applied and the computation never halts. In this way, an appearance checking rule is also correctly simulated. The simulation of a type 4 matrix is similar to that of a type 2 matrix. The rule $a \longrightarrow (a, out)$ can be applied to push the string out. If the string which leaves the system is purely over terminals, it is listed in the language. Hence, $RE \subseteq ERSP(nPri, nCoo)$.

5 Splicing Simple P Systems

In this section, we relate the idea of computing with membranes with another important area of natural computing, DNA Computing. We consider Simple P systems with objects in the form of strings and with the evolution rules based on splicing. First we define a splicing operation. Consider an alphabet V and two symbols $\#, \$$ not in V. A *splicing rule* over V is a string $r = u_1\#u_2\$u_3\#u_4$ where $u_1, u_2, u_3, u_4 \in V^*$. For such a rule r and for $x, y, w \in V^*$ we define $(x, y) \vdash_r w$ iff $x = x_1u_1u_2x_2, y = y_1u_3u_4y_2, w = x_1u_1u_4y_2$, for some $x_1, x_2, y_1, y_2 \in V^*$. We say that we splice the strings x and y at the sites u_1u_2 and u_3u_4 respectively. For clarity, we usually indicate by a vertical bar the place of splicing : $(x_1u_1|u_2x_2, y_1u_3|u_4y_2) \vdash x_1u_1u_4y_2$. Specifically, for each splicing rule $r = u_1\#u_2\$u_3\#u_4$ over a given alphabet V, we associate a string $z \in V^*$. For $x, y \in V^*$ we write $x \Longrightarrow_{(r,z)} y$ iff $(x, z) \vdash_r y$.
A splicing simple P system over a given alphabet V is a simple P system Π with strings as objects, with evolution rules given in the form $(r, z)tar$ where r is the splicing rule over V, $z \in V^*$, and tar is the target indication for the resulting string, one of $here, out, in\ j$. The indication $here$ is omitted usually. With respect to such a rule we define a relation $x \Longrightarrow_{(r,z)} y(tar)$as mentioned above. That is, if there is a string $x_1u_1u_2x_2$ in membrane i and if there is a rule $(x_1u_1\#u_2x_2\$y_1u_3\#u_4y_2, y_1u_3u_4y_2)in\ j$ where $y_1u_3u_4y_2$ is a string over V^*, and j is a membrane adjacent to i, then the string $x_1u_1u_4y_2$ moves to membrane j. Using this relation, we define the transition between configurations, taking into consideration also a possible priority among evolution rules. Here also, as in the case of rewriting simple P systems, we apply only one rule to a string, the parallelism refers to processing strings in all membranes simultaneously. We do not provide the membrane dissolving action again as it is not required for computational completeness. A computation is correctly finished in the same conditions as in the previous sections: no further move is possible. The language generated by Π consists of all strings over T^* sent out of the system at the end of a halting configuration. Note that a rewriting simple P system and splicing simple P system differ only in the evolution rules: in a rewriting system,

the evolution rules are rewriting rules, in a splicing system, the rules are splicing rules. The way the rules are applied and the the resultant of a computation are defined exactly in the same way for both the systems. We denote by $ESSP_m(\alpha)$ the language generated by splicing simple P systems with atmost m membranes, $\alpha \in \{Pri, nPri\}$.

Theorem 5.1 *The family $ESSP_3(nPri)$ contains non-regular languages and $ESSP_6(nPri)$ contains languages which are not in the family MAT.*

Proof : We first construct a splicing simple P system of degree 3 which contains non-regular languages. Consider $\Pi = (\{a, b, d, d_1, d_2, Z\}, \{a, b, d\}, [_1[_2[_3]_3]_2]_1, \{dabd\}, \lambda, \lambda, (R, \phi))$ where R consists of the rules : $r_1 : (da\#Z\$d\#a, daZ)in\ 2$, $r_2 : (\#Z\$d\#a, Z)out$, $r_3 : (b\#d\$Z\#d_1, Zd_1)in\ 3$, $r_4 : (b\#d_1\$Z\#bd_2, Zbd_2)out$, $r_5 : (b\#d_2\$Z\#d, Zd)out$. Initially, we have $dabd$ in the skin membrane. The possible rules which can be applied now are r_1 or r_2. The application of r_2 sends abd out of the system. If r_1 is used, the string goes to membrane 2 with an additional a. Now the applicable rules are r_2, r_3. r_3 changes the right end marker d to d_1 and the string is moved to membrane 3. Otherwise if r_2 is applied, we have in the skin, $aabd$ and the system halts as no more rules are applicable. In the former case, we can either apply r_4 by which we have the string $daabbd_2$ in membrane two; or r_2 by which $aabd_1$ comes to membrane 2. If the string present in membrane 2 is $aabd_1$, the only applicable rule is r_4, and this puts $aabbd_2$ in the skin, and application of r_5 pushes the string $aabbd$ out of the system. If on the other hand, the string present in membrane 2 is $daabbd_2$, rules r_2 or r_5 can be applied. Application of r_2 leaves the string $aabbd_2$ in the skin and as above, $aabbd$ leaves the system. r_5 puts the string $daabbd$ in the skin, from where $aabbd$ can leave the system by applying r_2. Proceeding in this way, the language generated by Π is $\{a^n b^n d \mid n \geq 1\}$.
Now we construct a splicing simple P system of degree 6 to show that the family $ESSP_6(nPri)$ contains languages outside the family MAT. The system $\Pi = (\{X, Y, Y', Y'', Z, a, b, c, c'\}$,
$\{a, Y\}, \{XabY\}, \lambda, \lambda, \lambda, \lambda, \lambda, [_1[_2[_3]_3]_2[_4[_5]_5]_4[_6]_6]_1, (R, \phi))$, where R consists of the rules $r_1 : (X\#Z\$Xa\#, XZ)in\ 2$, $r_2 : (\#Y\$Z\#aaY', ZaaY')in\ 3$, $r_3 : (\#Y'\$\#Y'', Y'')out$, $r_4 : (\#Y''\$\#Y, Y)out$, $r_5 : (X\#Z\$Xb\#, XZ)in\ 4$, $r_6 : (\#Y\$Z\#bY', ZbY')in\ 5$, $r_7 : (c\#Z\$Xb\#, cZ)in\ 6$, $r_8 : (c'\#a\$c\#a, c'a)out$, $r_9 : (\#a\$c'\#a, a)out$ generates the language $\{a^{2^n} Y \mid n \geq 1\}$. The system works as follows: Assume that we have a string of the form Xa^iba^jY in membrane one; initially we have i=1, j=0 . if $i \geq 1$, then we have to use the rule $X\#Z\$Xa\#$ and the string $Xa^{i-1}ba^jY$ is sent to membrane 2. The only applicable rule now is $\#Y\$Z\#aaY'$ and we get the string $Xa^{i-1}ba^{j+2}Y'$ in membrane 3. In this way, the number of a's is doubled every time the string goes to membrane 3. In the next two steps, we apply $\#Y'\$\#Y''$ and $\#Y''\$\#Y$ and we obtain the string $Xa^{i-1}ba^{j+2}Y$ in membrane one. In this way, we will eventually obtain $Xba^{j+2i}Y$ in membrane 1. Then if the rule $X\#Z\$Xb\#$ is applied, we obtain the string $Xa^{j+2i}Y$ in membrane 4. The only applicable rule now is $\#Y\$Z\#bY'$ which puts $Xa^{j+2i}bY'$ in membrane 5, and by applying $\#Y'\$\#Y''$, $\#Y''\$\#Y$, we obtain $Xa^{j+2i}bY$ in the skin membrane and the above process can be iterated. To terminate the above process, we apply to the string $Xba^{j+2i}Y$ in the

skin membrane the rule $c\#Z\$Xb\#$. Then we obtain the string $ca^{j+2i}Y$ in membrane 6. Applying $c'\#a\$c\#a$, the string $c'a^{j+2i}Y$ comes to the skin membrane. This string then leaves the system as $a^{j+2i}Y$ after $\#a\$c'\#a$ is applied to $c'a^{j+2i}Y$. Thus the language generated is $\{a^{2^n}Y \mid n \geq 1\}$.

Theorem 5.2 $RE \subseteq ESSP_7(nPri)$

Proof : Let G = (N, T, S, P) be a type-0 Chomsky grammar. Assume that $N \cup T = \{D_1, D_2, \ldots, D_n\}$ and take a further symbol B, also denoted by D_{n+1}. We construct the following splicing simple P system $\Pi = (V, T, \mu, \lambda, XBSY, \lambda, \lambda, \lambda, \lambda, \lambda, (R, \phi))$
$V = N \cup T \cup \{B, d, X, Y, Z, Z', X_j, Y_i, Y_i', Y_i'', Y_0''', Y_0^4, Y_0^5, X', X'', X''', X^4, \dagger \mid 1 \leq j \leq n, 0 \leq i \leq n\}$, $\mu = [_1\ [_2[_3[_4[_5]_5]_4]_3]_2\ [_6[_7]_7]_6\ ]_1$ and the rules are
$r_1 : (\#uY\$Z\#vY, ZvY)$ such that $u \longrightarrow v$ is a rule from P
$r_2 : (\#D_iY\$Z\#Y_i', ZY_i')out, 1 \leq i \leq n+1$; $r_3 : (X_iD_i\#Z\$X\#, X_iD_iZ)in\ 2, 1 \leq i \leq n+1$
$r_4 : (\#Y_i'\$Z\#Y_i, ZY_i)in\ 3, 0 \leq i \leq n+1$; $r_5 : (\#Y_i\$Z\#Y_{i-1}'', ZY_{i-1}'')out, 1 \leq i \leq n+1$
$r_6 : (\#Y_i''\$Z\#Y_i', ZY_i')out, 1 \leq i \leq n+1$; $r_7 : (X_{i-1}\#Z\$X_i\#, X_{i-1}Z)in\ 2, 2 \leq i \leq n+1$
$r_8 : (X\#Z\$X_1\#, XZ)in\ 6$; $r_9 : (\#Y_i'\$Z\#\dagger, Z\dagger)in\ 7, 1 \leq i \leq n+1$
$r_{10} : (\#Y_0'\$Z\#Y_0''', ZY_0''')in\ 7$; $r_{11} : (\#Y_0\$Z\#Y, ZY)in\ 4$
$r_{12} : (X'\#Z\$X\#, X'Z)in\ 5$; $r_{13} : (\dagger\#Z\$X_i\#, \dagger Z)in\ 5, 1 \leq i \leq n+1$
$r_{14} : (\dagger\#Z\$\dagger\#, \dagger Z)$; $r_{15} : (\#\dagger\$Z\#\dagger, Z\dagger)$
$r_{16} : (\#Y_0'''\$Z\#Y_0^4, ZY_0^4)out$; $r_{17} : (\#Y_0^4\$Z\#Y_0^5, ZY_0^5)out$
$r_{18} : (\#Y_0^5\$Z\#Y_0', ZY_0')in\ 2$; $r_{19} : (X''\#Z\$X'\#, X''Z)out$
$r_{20} : (X'''\#Z\$X''\#, X'''Z)out$; $r_{21} : (X\#Z\$X'''\#, XZ)out$
$r_{22} : (\#BY\$Z\#Z', ZZ')$; $r_{23} : (\#Z'\$Z\#d, Zd)out$
$r_{24} : (\#d\$Z\#, Z)in\ 6$; $r_{25} : (X^4D_i\#\$XD_i\#, X^4D_i)out,\ D_i \in T$
$r_{26} : (\#Z\$X^4\#, Z)out$
The system works as follows: In the initial configuration, we have the string $XBSY$ in membrane two, which introduces the axiom of G, together with a new symbol B and end markers X and Y. Assume that we have a string of the form XwY in membrane two. If we apply a splicing rule $\#uY\$Z\#vY$, then we simulate the use of a rule from P at the end of the string, $Xw'uY \Longrightarrow Xw'vY$, and this corresponds to $w'u \Longrightarrow w'v$ in G. The string remains in membrane 2. In the next step, we can either apply the above rule itself or perform a splicing $(Xw'|D_iY, Z|Y_i') \vdash Xw'Y_i'$. Then the string exits membrane two. In the skin membrane, if the rule $X_jD_j\#Z\$X\#$ is applied, we get a string $X_jD_jw'Y_i$ which is again passed to membrane 2. Here, we have to apply the rule $\#Y_i'\$Z\#Y_i$ and the string is passed to membrane 3 with Y_i' replaced by Y_i. In the next two steps, the only applicable rules are $\#Y_i\$Z\#Y_{i-1}''$ and $\#Y_{i-1}''\$Z\#Y_{i-1}'$ which decrements the subscript of the right end marker by one and the string is placed in the skin membrane. Now the rule $X_{i-1}\#Z\$X_i\#$ should be applied and the subscript of the left end marker is decreased by one and the string moves to membrane 2 and the process is repeated. When in the skin membrane we have the string $X_kD_jw'Y_0'$, it is passed to membrane 6 if k=1 using the rule $X\#Z\$X_1\#$ from where the rules $\#Y_0'\$Z\#Y_0''', \#Y_0'''\$Z\#Y_0^4, \#Y_0^4\$Z\#Y_0^5, \#Y_0^5\$Z\#Y_0', \#Y_0'\$Z\#Y_0, \#Y_0\$Z\#Y, X'\#Z\$X\#,$ $X''\#Z\$X'\#, X'''\#Z\$X''\#, X\#Z\$X'''\#$ take the string to membrane two as $XD_jw'Y$

passing through membranes 7, 6, 1, 2, 3, 4, 5, 4, 3, 2 in order. If $k \neq 1$, then we apply to $X_kD_jw'Y'_0$ in the skin membrane the rule $X_{i-1}\#Z\$X_i\#Z$ and we have the string $X_{k-1}D_jw'Y'_0$ in membrane 2, $k-1 \geq 1$. Here, the rule $\#Y'_0\$Z\#Y_0$ is applied and we have the string $X_{k-1}D_jw'Y_0$ in membrane 3. Then the rule $\#Y_0\$Z\#Y$ is applied and the string moves to membrane 4 with Y_0 replaced by Y. In membrane 4, the rule $\dagger\#Z\$X_{k-1}\#,\ k-1 \geq 1$ is applied and the computation never halts. (r_{14} can be applied forever)
Suppose that in the skin membrane we have the string $X_1D_jw'Y'_k$, with $k \geq 1$, then we apply r_8 and the string moves to membrane 6. Now if the rule $\#Y'_i\$Z\#\dagger$ is applied, from the next step the rule r_{15} can be applied forever. Consequently, in order to finish correctly the computation, the subscripts of the end markers have to reach the value zero at the same time, that is i = j. This means the symbol D_i which was cut from the right hand end of the string has been reproduced in the left end of the string. Note that the symbol B can be moved from one end of the string to the other like any symbol from $N \cup T$. In this way, the string is circularly permuted making possible the the simulation of rules of G in any position. If in Π we have generated the string Xw_1Bw_2Y then the string w_2w_1 is a sentential form of G, and conversely. To terminate, we apply r_{22} to the string in membrane 2. The right end marker and the symbol B are removed. In the next step, we apply r_{23} and the string is sent to membrane 1 with d as the right end marker. Then in the next three steps, the rules r_{24}, r_{25} and r_{26} are applied; r_{24} removes d, r_{25} replaces $XD_i, D_i \in T$ in the left end of the string by X^4D_i, and r_{26} removes X^4 and sends the string out of the system. If rules are applied in a different order from that stated above(this can happen since rules $r_1, r_2, r_5, r_{22}, r_{25}$ can be applied at any time; irrespective of which membrane the string is in), then either the system halts with no string going out or the strings leaving the system will not be listed in the language. Hence the language generated by Π consists of all strings over T^* generated by G.

6 Final Remarks

We have considered a new variant of super-cell systems, based on the natural modification in the way of applying a single set of rules, in comparison with the usual way of applying separate set of rules for each membrane. The minimum number of membranes required to get a characterization of RE using rewriting simple P systems of type (n Pri, n Coo) and whether there exists a splicing simple P system with lesser than seven membranes and no priorities which can generate recursively enumerable languages are problems to be pursued. It is also worthwhile to investigate whether this system can solve any hard problems.

References

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

[2] J. Dassow, Gh. Păun, On the power of membrane computing, *J. of Universal Computer Sci.*, 5, 2 (1999), 33–49 (www.iicm.edu/jucs).

[3] S. N. Krishna, R. Rama, A variant of P systems with active membranes: Solving NP-complete problems, *Romanian J. of Information Science and Technology*, 2, 4 (1999).

[4] S. N. Krishna, R. Rama, On Power of P systems based on sequentual and parallel rewriting *International J. of Computer Mathematics*, Vol 77 (1 or 2), 1 - 14, to appear.

[5] Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, to appear and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).

[6] Gh. Păun, Computing with membranes – A variant: P Systems with Polarized Membranes, *IJFOCS*, in press, and *Auckland University, CDMTCS Report* No 098, 1999.

[7] Gh. Păun, P systems with active membranes: Attacking NP complete problems, submitted 1999, and *Auckland University, CDMTCS Report* No 102, 1999.

[8] Gh.Păun, Computing with P Systems: Twenty Six Research Topics, Personal Communication

[9] Gh. Păun, G. Rozenberg, A. Salomaa, Membrane computing with external output, *Fundamenta Informaticae.*

[10] Gh. Păun, Y. Sakakibara, T. Yokomori, P systems on graphs of restricted forms, *IFIP Conf. on TCS: Exploring New Frontiers of Theoretical Informatics*, Sendai, Japan, 2000.

[11] Gh. Păun, T. Yokomori, Membrane computing based on splicing, *Preliminary Proc. of Fifth Intern. Meeting on DNA Based Computers* (E. Winfree, D. Gifford, eds.), MIT, June 1999, 213–227.

[12] Gh.Păun, S.Yu, On synchronization in P systems, *Fundamenta Informaticae*, 38, 4 (1999), 397–410.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 138 - 148.

Rational, Linear and Algebraic Languages of Multisets

Manfred Kudlek

Abstract

The theory of algebraic characterization of rational, linear and algebraic languages over an ω-complete semiring, defined by corresponding systems of equations, is applied for various underlying operations on multisets.

0. Introduction

For ω-complete semirings rational, linear and algebraic languages can be defined as solutions of corresponding systems of equations. These solutions are least fixed points which are limits starting with he empty sets. It can be shown that, similar to the well-known normal forms for regular, linear and context-free languages with catenation as underlying operation, normal forms for such systems also hold. Furthermore, corresponding grammars and trees (or better forests) can be constructed, too. If the nderlying operation is commutative, then regular, linear and algebraic languages coincide.

In part 1 the necessary definitions and results from the theory of ω-complete semirings are presented. In part 2 grammars, trees, and forests are constructed, and it is shown that they define languages identical to such defined as least fixed points. Following that normal forms are shown if a non-divisibity condition of the unit element is true. Finally, in part 3, several associative operations on multisets are presented.

It is also possible to define norms on multisets, fulfilling some monotonicity condition, such that iteration lemmata for multiset languages hold.

1. Systems of Equations

In this section the definitions of rational, linear and algebraic languages as least fixed points of corresponding systems of equations are introduced.

Let $\mathcal{M}$ be a monoid with binary operation $\circ$ and unit element $\mathbf{1}$, or with a binary operation $\circ : \mathcal{M} \times \mathcal{M} \to \mathcal{P}(\mathcal{M})$ with unit element $\mathbf{1}$, i.e. $\mathbf{1} \circ \alpha = \alpha \circ \mathbf{1} = \{\alpha\}$.

Extend $\circ$ to an associative operation $\circ : \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathcal{M}) \to \mathcal{P}(\mathcal{M})$, being distributive with union $\cup$ ($A \circ (B \cup C) = (A \circ B) \cup (A \circ C)$ and $(A \cup B) \circ C = (A \circ B) \cup (B \circ C)$), with unit element $\{\mathbf{1}\}$ ($\{\mathbf{1}\} \circ A = A \circ \{\mathbf{1}\} = A$), and zero element $\emptyset$, i.e. $\emptyset \circ A = A \circ \emptyset = \emptyset$.

Then $\mathcal{S} = (\mathcal{P}(\mathcal{M}), \cup, \circ, \emptyset, \{\mathbf{1}\})$ is an ω-complete semiring, i.e. if $A_i \subseteq A_{i+1}$ for $0 \leq i$ then $B \circ \bigcup_{i \geq 0} A_i = \bigcup_{i \geq 0}(B \circ A_i)$ and $(\bigcup_{i \geq 0} A_i) \circ B = \bigcup_{i \geq 0}(A_i \circ B)$.

Define also $A^{(0)} = \{\mathbf{1}\}$, $A^{(1)} = A, A^{(k+1)} = A \circ A^{(k)}$, $A^{\circ} = \bigcup_{k \geq 0} A^{(k)}$.

Let $\mathcal{X} = \{X_1, \dots, X_n\}$ be a set of variables such that $\mathcal{X} \cap \mathcal{M} = \emptyset$.

A *monomial* over $\mathcal{S}$ with variables in $\mathcal{X}$ is a finite string $A_1 \circ A_2 \circ \dots \circ A_k$, where $A_i \in \mathcal{X}$ or $A_i \subseteq \mathcal{M}, |A_i| < \infty, i = 1, \dots, k$. Without loss of generality, $A_i = \{\alpha_i\}$ with $\alpha_i \in \mathcal{M}$ suffices. The α_{ij} (or $\{\alpha_{ij}\}$) will be called *constants*. A *polynomial* $p(\underline{X})$ over $\mathcal{S}$ is a finite union of monomials where $\underline{X} = (X_1, \cdots, X_n)$.

In the following the symbol $\prod$ will be used to denote finite products with operation $\circ$:

$$\prod_{i=1}^{m} A_i = A_1 \circ \cdots \circ A_m$$

and the symbol $\sum$ to denote finite unions :

$$\sum_{i=1}^{n} A_i = \bigcup_{i=1}^{n} A_i = A_1 \cup \cdots \cup A_n \ .$$

A *system of equations* over $\mathcal{S}$ is a finite set of equations :

$\mathcal{E} := \{X_i = p_i(\underline{X}) \mid i = 1, \dots, n\}$, where $p_i(\underline{X})$ are polynomials. This will also be denoted by $\underline{X} = \underline{p}(\underline{X})$.

The *solution* of $\mathcal{E}$ is a n-tuple $\underline{L} = (L_1, \dots, L_n) \in \mathcal{P}(\mathcal{M})^n$, of sets over $\mathcal{M}$, and the n-tuple is minimal with this property, i.e. if $\underline{L'} = (L'_1, \dots, L'_n)$ is another n-tuple satisfying $\mathcal{E}$, then $\underline{L} \leq \underline{L'}$ (where the order is defined component- wise with respect to inclusion : $\underline{A} = (A_1, \cdots, A_n) \leq (B_1, \cdots, B_n) = \underline{B} \Leftrightarrow \forall_{i=1}^{n} : A_i \subseteq B_i$).

¿From the theory of semirings follows that any system of equations over $\mathcal{S}$ has a unique solution, and this is the least fixed point starting with

$\underline{X}^{(0)} = (X_1^{(0)}, \cdots, X_n^{(0)}) = (\emptyset, \cdots, \emptyset) = \underline{\emptyset}$, and $\underline{X}^{t+1} = \underline{p}(\underline{X}^{(t)})$

Then the following fact holds : $\underline{X}^{(t)} \leq \underline{X}^{(t+1)}$ for $0 \leq t$.

This is seen by induction and the property of the polynomial with respect to inclusion, as $\underline{\emptyset} \leq \underline{X}^{(1)}$ and $\underline{X}^{(t+1)} = \underline{p}(\underline{X}^{(t)}) \leq \underline{p}(\underline{X}^{(t+1)}) = \underline{X}^{(t+2)}$.

For the theory of semirings see [1, 4].

A general system of equations is called *algebraic*, *linear* if all monomials are of the form $A \circ X \circ B$ or A, and *rational* if they are of the form $X \circ A$ or A, with $A \subseteq M$ and $B \subseteq M$. Corresponding families of languages (solutions of such systems of equations) are denoted by $\underline{ALG(\circ)}$, $\underline{LIN(\circ)}$, and $\underline{RAT(\circ)}$. In the case $\circ$ is commutative then all families are identical : $\underline{ALG(\circ)} = \underline{LIN(\circ)} = \underline{RAT(\circ)}$.

Note that the algebraic case corresponds to context-free languages if $\circ$ is normal catenation.

Grammars

Interpreting an equation $X_i = p_i(\underline{X})$ as a set of rewriting productions $X_i \to m_{ij}$ with $m_{ij} \in M(X_i)$ where $M(X_i)$ denotes the set of monomials of $p_i(\underline{X})$, *regular, linear*, and *context-free* grammars $G_i = (\mathcal{X}, \mathcal{C}, X_i, P)$ using the operation $\circ$, can be defined. Here $\mathcal{C}$ stands for the set of all constants in the system of equations, and P for all productions defined as above. As the productions are *context-free* (*terminal*) derivation trees can also be defined. Note that the interior nodes are labelled by variables, and the leafs by constants from $\mathcal{C}$.

2 Normal Forms

In the following lemma forests of terminal trees are constructed representing approximations of the least fixed point, and it is shown that the stes of terminal derivation trees with respect to $\circ$ are equivalent.

Lemma 1 : *(Approximation of the least fixed point)*

Terminal trees for the approximation of the least fixed point and terminal derivation trees are *equivalent.*

Proof:

$\underline{X}^{(0)} = \underline{\emptyset}$, $\underline{X}^{(t+1)} = \underline{p}(\underline{X}^{(t)})$

Thus

$$X_i^{(t+1)} = \sum_j \prod_k X_{ijk}^{(t)} + \sum_j \{\alpha_{ij}\}$$

especially

$$X_i^{(0)} = \emptyset \ , \ X_i^{(1)} = \sum_j \{\alpha_{ij}\}$$

Construct forests $\mathcal{T}$ of terminal trees as follows :

$\mathcal{T}^{(1)}$ consists of all trees with roots $X_i^{(1)}$ and children (only leafs) $\{\alpha_{ij}\}$ with $1 \leq i \leq n$.

$\mathcal{T}^{(t+1)}$ is constructed from trees in $\mathcal{T}^{(1)}$ as the set of trees with roots $X_i^{(t+1)}$ and their children either $X_{ijk}^{(t)}$ being roots of trees from $\mathcal{T}^{(t)}$ or $\{\alpha_{ij}\}$.

Thus the set of frontiers of leafs of all trees in $\mathcal{T}^{(t)}$ with root $X_i^{(t)}$ is just the approximation $X_i^{(t)}$.

On the other hand, any terminal derivation tree for X_i is contained in $\mathcal{T}$. For this, interprete a deepest non-terminal vertex (i.e. with greatest distance from the root) as $X_j^{(1)}$ for some j, and the root as $X_i^{(t+1)}$ for some i. Then all non-terminal vertices get some step number s with $1 \leq s \leq t+1$.

□

Lemma 2 :

Any linear system of equations can be transformed, with additional variables, into another one where all monomials are of the form $X \circ \alpha$, $\alpha \circ X$, or α, and the new system has identical minimal solutions in the old variables.

Proof : Consider any monomial $\alpha \circ X \circ \beta$. Replace it by $\alpha \circ Y$, and add a new equation $Y = X \circ \beta$. Then it is obvious that the new system has identical solutions in the old variables.

□

In the following it will be shown that any algebraic system of equations can be transformed, with additional variables, into a system of equations where all monomials have the form $X \circ Y$ or $\{\alpha\}$, and the new system has identical minimal solutions for the old variables. To prove this some lemmata have to be shown first. For that the ω-complete semiring has to have the following

Property

Let $\mathcal{S} = (\mathcal{P}(\mathcal{M}), \emptyset, \mathbf{1}, \cup, \circ)$ be an ω-complete semiring where $\mathcal{M}$ is a monoid. $\mathcal{S}$ has property $(\otimes)$, if

$(\otimes)$ $\mathbf{1} \in A \circ B \Leftrightarrow (\mathbf{1} \in A \wedge \mathbf{1} \in B)$.

This property is some kind of *nondivisibility* of the unit.

Lemma 3 :

If $(\otimes)$ holds then

$$\mathbf{1} \in \prod_{i=1}^{k} A_i \;\Leftrightarrow\; \forall_{i=1}^{k} : \mathbf{1} \in A_i$$

Proof : $\Leftarrow$ is trivial.

$\Rightarrow$: $\forall_{i=1}^{k} : \mathbf{1} \in A_i$ implies $1 \in A_1 \wedge \forall_{i=2}^{k} : \mathbf{1} \in A_i$ by property $(\otimes)$, and then induction.

$\square$

Let $\mathcal{X} = \{X_1, \cdots, X_n\}$ be a set of variables. To each variable $X \in \mathcal{X}$ in an algebraic system of equations $\mathcal{E}$ there exists a set of monomials $M(X)$ such that $X = \sum_{m \in M(X)}$.

Lemma 4 : (*Separation of variables and constants*)

For any algebraic system of equations there exists another one, possibly with additional variables, having the same (partial) solution in the original variables, for which the following property holds :

if $X_i = \prod_{j=1}^{r(i)} m_{ij}$ then each monomial is either of the form $\prod_{k=1}^{s(ij)} X_{ijk}$ or $\{\alpha_{ij}\}$ (a constant).

Proof : If m_{ij} is not of that form and not a constant then $m_{ij} = \prod_{k=1}^{s(ij)} A_{ijk}$ with A_{ijk} either a variable or a constant β_{ijk}. Replace each constant β_{ijk} in it by a new variable Y_{ijk}, and add a new equation $Y_{ijk} = \{\beta_{ijk}\}$.

Trivially, the new system of equations has the same solution in the original variables.

$\square$

Lemma 5 : (*Removal of* $\{\mathbf{1}\}$)

To each algebraic sysytem of equations there exists another one with the same set of variables such that no monomial has the form $\mathbf{1}$ and the solutions are $L_i - \{\mathbf{1}\}$ if L_i are the solutions the old system.

Proof :

Let $\mathcal{Y}$ be a set of variables and $\mathcal{F}(\mathcal{Y})$ the set of all (formal) terms on $\mathcal{Y}$ with operation $\circ$.

Define inductively

$\mathcal{Y}_1 = \{X \in \mathcal{X} \mid \mathbf{1} \in M(X)\}$, $\mathcal{Y}_{i+1} = \mathcal{Y}_i \cup \{X \in \mathcal{X} \mid \exists m \in \mathcal{F}(\mathcal{Y}_i) : m \in M(X)\}$

Note that all monomials m consist only of variables.

Trivially $\mathcal{Y}_i \subseteq \mathcal{Y}_{i+1}$, and therefore there exists a k with $\mathcal{Y}_k = \mathcal{Y}_{k+j} = \mathcal{Y}$ for all $0 \leq j$ since $\mathcal{X}$ is finite.

The following fact holds :

$\{\mathbf{1}\} \subseteq X \Leftrightarrow X \in \mathcal{Y}$.

$\Leftarrow$) If $X \in \mathcal{Y}$ then $\mathbf{1} \in X$ is seen by induction. Trivially, if $X \in \mathcal{Y}_1$ then $\mathbf{1} \in M(X)$ and therefore $\mathbf{1} \in X$. Assume $\mathbf{1} \in X$ for all $X \in \mathcal{Y}_j$ for $1 \leq j$. If $X \in \mathcal{Y}_{j+1}$ then by definition there exists a monomial $m \in \mathcal{F}(\mathcal{Y}_j)$ such that $m \in M(X)$. Therefore $\mathbf{1} \in X$.

$\Rightarrow$) Let $X = X_i$. $\mathbf{1} \in X_i$ implies $\{\mathbf{1}\} \subseteq X_i^{(t)}$ for some $t \geq 1$. Let t be minimal, i.e. $\mathbf{1} \notin X_i^s$ for $s < t$. If $t = 1$ then $1 \in M(X_i)$ and therefore $X_i \in \mathcal{Y}_1 \subseteq \mathcal{Y}$.

Let $t > 1$. If $\mathbf{1} \in M(X_i)$ then again $X_i \in \mathcal{Y}_1 \subseteq \mathcal{Y}$. By assumption for t $\mathbf{1} \notin M(X_i)$. Then $\{\mathbf{1}\} \subseteq Y_1^{(t-1)} \circ \cdots \circ Y_r^{(t-1)} = m_1 \in M(X_i)$. Property ($\otimes$) implies $\{\mathbf{1}\} \subseteq Y_j^{(t-1)}$ for $1 \leq j \leq r$. Put Y_j into the set $\mathcal{Z}$ if $\mathbf{1} \in M(Y_j)$, and repeat the procedure for all remaining $Y_j^{(t-1)}$ with $\mathbf{1} \notin M(Y_j)$. The procedure must terminate for some $Y_k^{(1)}$ for which $\mathbf{1} \in M(Y_k)$, yielding a set of variables $\mathcal{Z}$ with $\mathbf{1} \in M(Y)$ for $Y \in \mathcal{Z}$. Therefore $\mathcal{Z} \subseteq \mathcal{Y}$. By the construction there exists a $m \in M(X_i)$ with $m \in \mathcal{F}(\mathcal{Z}) \subseteq \mathcal{F}(\mathcal{Y})$. Obviously, $X_i \in \mathcal{Y}$.

Now construct a new system of equations $\mathcal{E}'$ in which in all monomials m_{ij} none or more variables $Y_j \in \mathcal{Y}$ are deleted, such that the new monomials $m'_{ij} \neq \{\mathbf{1}\}$.

Then the system $\mathcal{E}'$ has the solutions $L_i - \{\mathbf{1}\}$

□

Lemma 6

To each algebraic system of equations there exists another one with additional variables X'_i for each old X_i such that the monomials in $p'_i(\underline{X, X'})$ are either of the form $\{\mathbf{1}\}$ or don't contain X'_j. The solutions of the new system for the new variables X'_i are $L'_i = L_i$.

Proof :

By Lemma 5 let $\mathcal{E}'$ be a system of equations with $L'_i = L_i - \{\mathbf{1}\}$.

Construct a new system $\mathcal{E}''$ in which for each variable X_i a new one X'_i is defined. Let $p_i(\underline{X, X'}) = p_i(\underline{X})$ for X_i and define $p'_i(\underline{X, X'}) = \{\mathbf{1}\} + p_i(\underline{X})$ if $\mathbf{1} \in L_i$, and in case $\mathbf{1} \notin L_i$ $p'_i(\underline{X, X'}) = p_i(\underline{X})$. Then the solutions for the new variables are $L'_i = L_i$.

□

Lemma 7 : (*Removal of monomials of the form* Y)

To each algebraic system of equations there exists another one with the same variables such that no monomial is of the form Y and the solutions is identical to the old one.

Proof :

Assume that the system is already in the form according to lemmata 4, 5, and 6.

Construct inductively sets of variables for $X \in \mathcal{X}$:

$\mathcal{Y}_1(X) = \{X\}$

$\mathcal{Y}_{j+1}(X) = \mathcal{Y}_j(X) \cup \{Y \in \mathcal{X} \mid \exists Z \in \mathcal{Y}_j(X) : Y \in M(X)\}$

Since $\mathcal{X}$ is finite there exists a k with $\mathcal{Y}_k(X) = \mathcal{Y}_{k+j}(X) = \mathcal{Y}(X)$ for $j \geq 0$.

Obviously, the following fact holds : $Y \subseteq X \Leftrightarrow Y \in \mathcal{Y}(X)$.

Now construct the new system by taking all monomials which are constants and consider all monomials $m = Y_1 \circ \cdots \circ Y_k \in M(X)$ with $k \geq 2$. Construct the new monomials $m' = Z_i \circ \cdots \circ Z_k \in M(Y)$ with $X \in \mathcal{Y}(Y)$ and $Z_j \in \mathcal{Y}(Y_j)$.

Then $L'_i = L_i$.

□

Lemma 8 (*Normal form*)

To each algebraic system there exists another one with additional variables such that all monomials have only the forms $\mathbf{1} \in M(X)$ (then no other monomial contains X), or $Y \circ Z$, or $\{\alpha\}$ with $\alpha \neq \mathbf{1}$n and the solutions for the old variables are identical.

Proof :

Assume that the system of equations has the form according to the previous lemmata.

Consider an arbitrary monomial $m = Y_1 \circ \cdots \circ Y_k \in M(X)$ with $k \geq 2$. Replace it by $Y_1 \circ Z_1 \in M(X)$ and the new equations $Z_1 = Y_2 \circ Z_2, \cdots, Z_{k-2} = Y_{k-1} \circ Y_k$.

Then the new system of equations obviously has the same solutions in the old variables.

□

4 Multisets

Let $\Sigma = \{a_1, \cdots, a_n\}$ be an alphabet.

A *multiset* over Σ will either be denoted by $x = \langle \mu_x(a_1) \cdot a_1, \cdots, \mu_x(a_n) \cdot a_n) \rangle$ where $\mu_x(a_i)$ is the multiplicity of a_i, or as a vector $x = (\mu_x(a_1), \cdots, \mu_x(a_n)) \in I\!N^n$. Let the set of multisets over Σ be denoted by $\mathbf{M}(\Sigma)$.

If x is a multiset define $\sigma(x) = \sum_{i=1}^{n} \mu_x(a_i)$ as its *norm* or *length*.

Write $\xi \in x$ if $\mu_x(\xi) > 0$.

To be more general, instead of a finite alphabet Σ an infinite set may be considered, like Γ^* or $I\!N^k$ where Γ is a finite alphabet.

A multiset is then denoted by $x = \langle \mu(a_i) \cdot a_i \mid i \geq 0 \rangle$ with $a_i \in \Gamma^*$ (or $a_i \in I\!N^k$) and $\sum_{i=0}^{\infty} \mu_x(a_i) < \infty$.

For two multisets $x = \langle \mu_x(a_i) \cdot a_i \mid i \geq 0 \rangle$ and $y = \langle \mu_y(a_i) \cdot a_i \mid i \geq 0 \rangle$ define $x \subseteq y$ iff $\forall i \geq 0 : \mu_x(a_i) \leq \mu_y(a_i)$. Analogously, define $z = x \cup y$ by $\mu_z(a_i) = \mu_x(a_i) + \mu_y(a_i)$ for $i \geq 0$, and $z = x - y$ by $\mu_z(a_i) = max(0, \mu_x(a_i) - \mu_y(a_i))$.

Example 1 : (*Vector Addition System*)

Let n be fixed and consider $M_1 = \mathbb{N}^n$ with $\mathbf{0} = (0, \cdots, 0) \in \mathbb{N}^n$. Then the structure $\mathcal{M}_1 = (M_1, +, \mathbf{0})$ is a commutative monoid, and $\mathcal{S}_1 = (\mathcal{P}(M_1), \cup, +, \emptyset, \mathbf{0})$ a commutative ω-complete semiring.

Define

$$A + B = \bigcup_{a \in A, b \in B} (a + b)$$

$\sigma(A) = max\{\sigma(m) \mid m \in A\}$ with $\sigma(\emptyset) = \sigma(\{0\}) = 0$ defines a norm on $\mathcal{S}_1$.

□

Example 2 : (*Tensor Product*)

Consider

$$M_2 = \bigcup_{k=0}^{\infty} \mathbb{N}^k - \bigcup_{k=1}^{\infty} \{\mathbf{0}\}^k$$

($\mathbb{N}^0 = \{\mathbf{1}\}$ where $\mathbf{1}$ is considered as a unit element).

If $x = (x_1, \cdots, x_r) \in \mathbb{N}^r - \{0\}^r$, $y = (y_1, \cdots, y_s) \in \mathbb{N}^s - \{0\}^s$ define
$x \otimes y = (x_1 \cdot y, \cdots, x_r \cdot y) = (x_1y_1, \cdots, x_1y_s, \cdots, x_ry_1, \cdots, x_ry_s) \in \mathbb{N}^{r \cdot s} - \{0\}^{r \cdot s}$.
$\otimes$ is an associative operation since with $z = (z_1, \cdots, z_t) \in \mathbb{N}^t - \{0\}^t$
$(x \otimes y) \otimes z = ((x_1y_1) \cdot z, \cdots, (x_1y_s) \cdot z, \cdots, (x_ry_1) \cdot z, \cdots, (x_ry_s) \cdot z) =$
$(x_1y_1z_1, \cdots, x_1y_1z_t, \cdots, x_1y_sz_1, \cdots, x_1y_sz_t, \cdots, x_ry_1z_1, \cdots, x_ry_1z_t,$
$\cdots, x_ry_sz_1, \cdots, x_ry_sz_t)$
and
$x \otimes (y \otimes z) = (x_1 \cdot (y \otimes z), \cdots, x_r \cdot (y \otimes z)) =$
$(x_1y_1z_1, \cdots, x_1y_1z_t, \cdots, x_1y_sz_1, \cdots, x_1y_sz_t, \cdots, x_ry_1z_1, \cdots, x_ry_1z_t,$
$\cdots, x_ry_sz_1, \cdots, x_ry_sz_t)$.
Define $\mathbf{1} \otimes x = x \otimes \mathbf{1} = x$.
Then $\mathcal{M}_2 = (M_2, \otimes, \mathbf{1})$ is a monoid, and by extending $\otimes$ to $\mathcal{P}(M_2)$ follows that $\mathcal{S}_2 = (\mathcal{P}(M_2), \cup, \otimes, \emptyset, \{\mathbf{1}\})$ is an ω-complete semiring.
With σ as in Example 1 ($\sigma(\mathbf{1}) = 1$) follows $\sigma(A), \sigma(B) \leq \sigma(A \otimes B) \leq \sigma(A) \cdot \sigma(B)$.
With $\tau(x) = 1 + \lceil log_2(\sigma(x)) \rceil$ for $x \neq \mathbf{1}$ and $\tau(\mathbf{1}) = 0$ a usual norm is defined with
$\tau(A), \tau(B) \leq \tau(A) \otimes \tau(B) \leq \tau(A) + \tau(B)$.

□

Note that all $x \in \mathbb{N}^p$ with p a prime number are also prime with respect to $\otimes$.

Example 3 :

Consider

$$M_3 = \bigcup_{k=0}^{\infty} \mathbb{N}^{|\Sigma|^k} - \bigcup_{k=1}^{\infty} \{0\}^{|\Sigma|^k}$$

($\mathbb{N}^0 = \{\mathbf{1}\}$ where $\mathbf{1}$ is considered as a unit element).

Interprete $x \in M_3$ as a multiset representing the multiplicities of words of length k in lexicographical order. An operation $\odot : M_3 \times M_3 \to M_3$ is defined in the following way.

$$x \odot y = \langle \xi \cdot \eta \mid \xi \in x, \eta \in y \rangle \quad , \; \mathbf{1} \odot x = x \odot \mathbf{1} = x \; .$$

respecting all multiplicities, and where $\cdot$ is catenation.
Examples :
$\langle a, a, b \rangle \odot \langle aa, ba \rangle = \langle aaa, aaa, aba, aba, baa, bba \rangle$
or in other notation $(2,1) \odot (1,0,1,0) = (2,0,2,0,1,0,1,0)$.
$\langle a, a, b \rangle \odot \langle ab, ba \rangle = \langle aab, aab, bab, aba, aba, bba \rangle$
or in other notation $(2,1) \odot (0,1,1,0) = (0,2,2,0,0,1,1,0)$.
$\odot$ is an associative operation since
$(x \odot y) \odot z = \langle \xi \cdot \eta \cdot \zeta \mid \xi \in x, \eta \in y, \zeta \in z \rangle = x \odot (y \odot z)$.
Thus, $\mathcal{M}_3 = (M_3, \odot, \mathbf{1})$ is a monoid.
Extending $\odot$ to $\mathcal{P}(M_3)$ gives an ω-complete semiring $\mathcal{S}_3 = (\mathcal{P}(M_3), \cup, \odot, \emptyset, \{\mathbf{1}\})$.
$\square$

Example 4 :
In this example the elements of two multisets may combine or not.
Consider again as in Example 3

$$M_4 = \bigcup_{k=1}^{\infty} I\!N^{|\Sigma|^k} - \bigcup_{k=1}^{\infty} \{0\}^{|\Sigma|^k}$$

($I\!N^0 = \{\mathbf{1}\}$ where $\mathbf{1}$ is considered as a unit element).
With $\odot$ as in Example 3 define an operation $\otimes : M_4 \times M_4 \to \mathcal{P}(M_4)$ by

$$x \otimes y = \{x \odot y\} \cup \{x\} \cup \{y\} \quad , \; \mathbf{1} \otimes x = x \otimes \mathbf{1} = \{x\} \; .$$

$\otimes$ is an associative operation since
$(A \otimes B) \otimes C = (A \odot B \cup A \cup B) \otimes C = A \odot B \odot C \cup A \odot C \cup B \odot C \cup A \odot B \cup A \cup B$
$A \otimes (B \otimes C) = A \otimes (B \odot C \cup B \cup C) = A \odot B \odot C \cup A \odot C \cup A \odot C \cup A \cup B \odot C \cup B \cup C$
and therefore $(A \otimes B) \otimes C = A \otimes (B \otimes C)$.
Extending $\otimes$ to $\mathcal{P}(M_4)$ gives a monoid $\mathcal{M}_4 = (\mathcal{P}(M_4), \otimes, \{\mathbf{1}\})$ and an ω-complete semi- ring $\mathcal{S}_4 = (\mathcal{P}(M_4), \cup, \otimes, \emptyset, \{\mathbf{1}\})$.
Example :
$\langle a, a, b \rangle \otimes \langle ab, ba \rangle = \{\langle aab, aab, aba, aba, bab, bba \rangle\} \cup \{\langle a, a, b \rangle\} \cup \{\langle ab, ba \rangle\}$,
or in other notation
$(2,1) \otimes (0,1,1,0) = \{(0,2,2,0,0,1,1,0), (0,1,1,0), (2,1)\}$.
$\square$

Example 5 :
Consider again

$$M_5 = \bigcup_{k=0}^{\infty} I\!N^{|\Sigma|^k} - \bigcup_{k=1}^{\infty} \{0\}^{|\Sigma|^k}$$

($I\!N^0 = \{\mathbf{1}\}$ where $\mathbf{1}$ is considered as a unit element).

An operation $\odot : M_5 \times M_5 \to \mathcal{P}(M_5)$ is defined in the following way.

$$x \odot y = \langle \xi \sqcup\!\sqcup \eta \mid \xi \in x, \eta \in y \rangle \quad , \ \mathbf{1} \odot x = x \odot \mathbf{1} = x \ .$$

respecting all multiplicities, and where $\sqcup\!\sqcup$ is the shuffle operation.

Examples :

$\langle a, a, b\rangle \odot \langle aa, ba\rangle = \langle a \sqcup\!\sqcup aa, a \sqcup\!\sqcup ba, a \sqcup\!\sqcup aa, a \sqcup\!\sqcup ba, b \sqcup\!\sqcup aa, b \sqcup\!\sqcup ba\rangle$
$= \langle aaa, aba, baa, aaa, aba, baa, baa, aba, aab, bba, bab\rangle$
or in other notation $(2, 1) \odot (1, 0, 1, 0) = (2, 1, 3, 0, 3, 1, 1, 0)$.

$\langle a, a, b\rangle \odot \langle ab, ba\rangle = \langle a \sqcup\!\sqcup ab, a \sqcup\!\sqcup ba, a \sqcup\!\sqcup ab, a \sqcup\!\sqcup ba, b \sqcup\!\sqcup ab, b \sqcup\!\sqcup ba\rangle$
$= \langle aab, aba, aba, baa, aab, aba, aba, baa, bab, abb, bba, bab\rangle$
or in other notation $(2, 1) \odot (0, 1, 1, 0) = (0, 2, 4, 1, 2, 2, 1, 0)$.

$\odot$ is an associative operation since
$(x \odot y) \odot z = \langle \xi \sqcup\!\sqcup \eta \sqcup\!\sqcup \zeta \mid \xi \in x, \eta \in y, \zeta \in z\rangle = x \odot (y \odot z)$.

$\odot$ is a commutative operation since $x \odot y = \langle \xi \sqcup\!\sqcup \eta \mid \xi \in x, \eta \in y\rangle = y \odot x$.

Extending $\odot$ to $\mathcal{P}(M_3)$ gives a monoid $\mathcal{M}_5 = (\mathcal{P}(M_5), \odot, \{\mathbf{1}\})$ and an ω-complete semi- ring $\mathcal{S}_3 = (\mathcal{P}(M_3), \cup, \odot, \emptyset, \{\mathbf{1}\})$.

□

Example 6 :

In this example the elements of two multisets may combine or not.

Consider again as in Example 5

$$M_6 = \bigcup_{k=1}^{\infty} I\!N^{|\Sigma|^k} - \bigcup_{k=1}^{\infty} \{0\}^{|\Sigma|^k}$$

($I\!N^0 = \{\mathbf{1}\}$ where $\mathbf{1}$ is considered as a unit element).

With $\odot$ as in Example 5 define an operation $\otimes : M_6 \times M_6 \to \mathcal{P}(M_6)$ by

$$x \otimes y = \{x \sqcup\!\sqcup y\} \cup \{x\} \cup \{y\} \quad , \ \mathbf{1} \otimes x = x \otimes \mathbf{1} = \{x\} \ .$$

$\otimes$ is an associative operation since
$(A \otimes B) \otimes C = (A \sqcup\!\sqcup B \cup A \cup B) \otimes C = A \sqcup\!\sqcup B \sqcup\!\sqcup C \cup A \sqcup\!\sqcup C \cup B \sqcup\!\sqcup C \cup A \sqcup\!\sqcup B \cup A \cup B \cup C$
$A \otimes (B \otimes C) = A \otimes (B \sqcup\!\sqcup C \cup B \cup C) = A \sqcup\!\sqcup B \sqcup\!\sqcup C \cup A \sqcup\!\sqcup B \cup A \sqcup\!\sqcup C \cup A \cup B \sqcup\!\sqcup C \cup B \cup C$
and therefore $(A \otimes B) \otimes C = A \otimes (B \otimes C)$.

$\otimes$ is a commutative operation since $A \otimes B = A \sqcup\!\sqcup B \cup A \cup B = B \otimes A$.

Extending $\otimes$ to $\mathcal{P}(M_6)$ gives a monoid $\mathcal{M}_6 = (\mathcal{P}(M_6), \otimes, \{\mathbf{1}\})$ and an ω-complete semi- ring $\mathcal{S}_6 = (\mathcal{P}(M_6), \cup, \otimes, \emptyset, \{\mathbf{1}\})$.

□

Example 7 :

In this example multisets of vectors (multisets) on $I\!N^k$ for fixed k are considered. Let $M_7 = \mathbf{M}(I\!N^k)$.

Writing $\langle m_i \mid 1 \le i \le r\rangle$ for $\langle m_1, \cdots, m_r\rangle$, where some of the m_i may be identical, an operation $\odot : M_7 \times M_7 \to M_7$ is defined by

$\langle m_i \mid 1 \leq i \leq r\rangle \odot \langle m_j \mid 1 \leq j \leq s\rangle = \langle m_i + m_j \mid 1 \leq i \leq r, 1 \leq j \leq s\rangle$.

The unit element is $\langle \mathbf{0}\rangle \in \mathbf{M}(I\!N^k)$.

Trivially, $\odot$ is a commutative and associative operation, and therefore $\mathcal{M}_7 = (M_7, \odot, \langle \mathbf{0}\rangle)$ is a commutative monoid, and $\mathcal{S}_7 = (\mathcal{P}(M_7), \cup, \odot, \emptyset, \{\langle \mathbf{0}\rangle\})$ an ω-complete semiring.

Example :

$\langle(1,1),(1,1),(2,0)\rangle \odot \langle(0,2),(1,1)\rangle = \langle(1,3),(1,3),(2,2),(2,2),(2,2),(3,1)\rangle$.

□

Example 8 :

In this example again multisets of vectors (multisets) on $I\!N^k$ for fixed k are considered. Let $M_8 = \mathbf{M}(I\!N^k)$.

Define an operation $\otimes : M_8 \times M_8 \to \mathcal{P}(M_8)$ in the following way.

Let $x, y \in M_8$. Consider the multiset partitions $x = x_{12} \cup x_1$ and $y = y_{12} \cup y_2$ with $|x_{12}| = |y_{12}| = p$. Order x_{12} and y_{12}, i.e. $x_{12} = \langle \xi_1, \cdots, \xi_p\rangle$ and $y_{12} = \langle \eta_1, \cdots, \eta_p\rangle$, and define $x_{12} + y_{12} = \langle \xi_1 + \eta_1, \cdots, \xi_p + \eta_p\rangle$.

Then let $(x_{12} + y_{12}) \cup x_1 \cup y_2 \in x \otimes y$ for all partitions and all orderings.

The unit element is $\langle \mathbf{0}\rangle$ with $\mathbf{0} \in I\!N^k$.

Trivially, $\otimes$ is a commutative operation. $\otimes$ is also associative. To show that consider the following partitions.

$x = x_{123} \cup x_{12} \cup x_{13} \cup x_1$, $y = y_{123} \cup y_{12} \cup y_{23} \cup y_2$, $z = z_{123} \cup z_{13} \cup z_{23} \cup z_3$

with $|x_{123}| = |y_{123}| = |z_{123}|$, $|x_{12}| = |y_{12}|$, $|x_{13}| = |z_{13}|$, $|y_{23}| = |z_{23}|$,

such that $\tilde{x}_{12} = x_{123} \cup x_{12}$, $\tilde{x}_1 = x_{13} \cup x_1$, $\tilde{y}_{12} = y_{123} \cup y_{12}$, $\tilde{y}_2 = y_{23} \cup y_2$ for $x \otimes y$

and $\hat{y}_{23} = y_{123} \cup y_{23}$, $\hat{y}_2 = y_{12} \cup y_2$, $\hat{z}_{23} = z_{123} \cup z_{23}$, $\hat{z}_3 = z_{13} \cup z_3$ for $y \otimes z$.

Then $x \otimes y = (\tilde{x}_{12} + \tilde{y}_{12}) \cup \tilde{x}_1 \cup \tilde{y}_2$

$= ((x_{123} \cup x_{12}) + (y_{123} \cup y_{12})) \cup (x_{13} \cup x_1) \cup (y_{23} \cup y_2)$

$= (x_{123} + y_{123}) \cup (x_{12} + y_{12}) \cup x_{13} \cup y_{23} \cup x_1 \cup y_2 \in x \otimes y$ and

$(x_{123}+y_{123}+z_{123}) \cup (x_{13}+z_{13}) \cup (y_{23}+z_{23}) \cup (x_{12}+y_{12}) \cup x_1 \cup y_2 \cup z_3 \in (x \otimes y) \otimes z$.

Using the same partitions and orderings implies

$y \otimes z = (\hat{y}_{23} + \hat{z}_{23}) \cup \hat{y}_2 \cup \hat{z}_3$

$= ((y_{123} \cup y_{23}) + (z_{123} + z_{23})) \cup (y_{12} \cup y_2) \cup (z_{13} \cup z_3)$

$= (y_{123} + z_{123}) \cup (y_{23} + z_{23}) \cup y_{12} \cup z_{13} \cup y_2 \cup z_3 \in y \otimes z$ and

$(x_{123}+y_{123}+z_{123}) \cup (x_{13}+z_{13}) \cup (y_{23}+z_{23}) \cup (x_{12}+y_{12}) \cup x_1 \cup y_2 \cup z_3 \in x \otimes (y \otimes z)$.

The opposite is shown in a similar way. Thus $\mathcal{M}_8 = (\mathcal{P}(M_8), \otimes, \{\langle \mathbf{0}\rangle\})$ is a commutative monoid, and $\mathcal{S}_8 = (\mathcal{P}(M_8), \cup, \otimes, \emptyset, \{\langle \mathbf{0}\rangle\})$ a commutative ω-complete semiring.

Example : $\langle(0,1),(1,0)\rangle \otimes \langle(0,1),(0,1),(1,0),(1,1)\rangle$

$= \{\langle(0,1),(0,1),(0,1),(1,0),(1,0),(1,1)\rangle\}$ ($x_{12} = \emptyset$)

$\cup \{\langle(0,1),(0,2),(1,0),(1,0),(1,1)\rangle,$

$\langle(0,1),(0,1),(1,0),(1,1),(1,1)\rangle,$

$\langle(0,1),(0,1),(1,0),(1,0),(1,2)\rangle\}$ ($x_{12} = \langle(0,1)\rangle$)

$\cup \{\langle(0,1),(0,1),(1,0),(1,1),(1,1)\rangle,$

$\langle(0,1),(0,1),(0,1),(1,1),(2,0)\rangle,$

$\langle(0,1),(0,1),(0,1),(1,0),(2,1)\rangle\}$ ($x_{12} = \langle(1,0)\rangle$)

$\cup\{\langle(0,2),(1,0),(1,1),(1,1)\rangle,$

$$\begin{aligned}
&\langle(0,1),(0,2),(1,1),(2,0)\rangle,\\
&\langle(0,1),(1,1),(1,1),(1,1)\rangle,\\
&\langle(0,1),(0,2),(1,0),(2,1)\rangle,\\
&\langle(0,1),(1,0),(1,1),(1,2)\rangle\}\quad(\ x_{12}=\langle(0,1),(1,0)\rangle\ ).
\end{aligned}$$

$\square$

References

[1] J. S. Golan : *The Theory of Semirings with Application in Mathematics and Theoretical Computer Science.* Longman Scientific and Technical, 1992.

[2] M. Kudlek : *Generalized Iteration Lemmata.* PU.M.A., Vol. 6 No. 2, 211-216, 1995.

[3] M. Kudlek : *Iteration Lemmata for Certain Classes of Word, Trace and Graph Languages.* Fundamenta Informaticae, Vol. 34, 249-264, 1999.

[4] W. Kuich, A. Salomaa : *Semirings, Automata, Languages.* EATCS Monographs on Theoretical Computer Science 5, Springer, Berlin, 1986.

[5] A. Salomaa : *Formal Languages.*

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 149 - 158.

Toward FMT (Formal Macroset Theory)

Manfred KUDLEK
Fachbereich Informatik, Universität Hamburg
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany
E-mail: `kudlek@informatik.uni-hamburg.de`

Carlos MARTÍN-VIDE
Research Group on Mathematical Linguistics
Rovira i Virgili University, Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
E-mail: `cmv@astor.urv.es`

Gheorghe PĂUN[1]
Institute of Mathematics of the Romanian Academy
PO Box 1-764, 70700 Bucureşti, Romania
E-mail: `gpaun@imar.ro`

Abstract. A *macroset* is a (finite or infinite) set of multisets over a finite alphabet. We introduce a Chomsky-like hierarchy of multiset rewriting devices which, therefore, generate macrosets. Some results are proved about the power of these devices and some open problems are formulated. We also present an algebraic characterization of some of the macroset families as least fixed point solutions of algebraic systems of equations.

1 Introduction

In the last years, the idea of multiset processing has appeared more and more frequently in various domains: nondeterministic programming [1], the chemical abstract machine [6], DNA computing [25], membrane computing (P systems) [22], [23] (see a current bibliography of the area in [24]). Multisets also appear in logic [4], linguistics [7], [12], artificial life [27], etc. The thesis [29] is entirely devoted to mathematically formalizing multisets. Several papers have considered fuzzy variants of multisets ([21], [18], [32]), while [10] and [5] deal with *pomsets* (partially ordered multisets).

A multiset over a given set of *objects* corresponds to a string over the alphabet naming those objects. To a language it corresponds a sets of multisets, called here a *macroset*.

Observing that P system theory is devoted to a generative approach to multiset processing in a distributed way, in [24] one formulates the problem of producing a non-distributed theory of multiset generating, like Chomsky grammar theory. We systematically address here this question, by considering a natural translation of notions related to sequential rewriting of strings to rewriting of multisets. A Chomksy-like hierarchy of *multiset grammars* is obtained, but with some relationships between classes of macrosets which are different from those among the corresponding families of languages.

[1]Research supported by the Direcció General de Recerca, Generalitat de Catalunya (PIV).

In this framework, an important question remains open: is erasing important in the case of non-context-free macroset grammars? This corresponds to the problem whether or not the Parikh sets of string context-free matrix languages generated with and without using erasing rules are the same, which, in turn, is related to he problem still open in the regulated rewriting area whether or not erasing rules increase the power of string matrix grammars (without appearance checking).

It should be noted that in [27], [28] one already considers an abstract rewriting multiset system (ARMS, for short), which corresponds to our arbitrary multiset grammars, without introducing particular (Chomsky-like) variants; the ARMS are mainly used as a formal framework for simulating the development of populations in artificial life-like questions. Moreover, languages of *multisets* (or *bags*), also called *commutative* languages, and language families (trios, semi AFL's, AFL's) of commutative languages, have been considered in [8], [15], [16], [17], [13]. None of these papers proposes a Chomsky-like hierarchy of multiset processing grammars.

2 Multiset Grammars

All formal language theory prerequisites we use can be found in [26].

Let V be a finite alphabet; its elements are called both *symbols* and *objects*. The set of strings over V is denoted by V^*; the empty string is denoted by λ, the length of $x \in V^*$ is denoted by $|x|$, and the number of occurrences of a symbol $a \in V$ in a string $x \in V^*$ is denoted by $|x|_a$.

A multiset over an alphabet V is a mapping $\mu : V \longrightarrow \mathbf{N}$. We denote by $V^{\#}$ the set of all multisets over V. We can represent the multiset μ as a vector, $\mu(V) = (\mu(a_1), \ldots, \mu(a_n))$, for $V = \{a_1, \ldots, a_n\}$. Note that the ordering of the objects in V is relevant. The multiset μ can also be represented by any permutation of the string $w_\mu = a_1^{\mu(a_1)} \ldots a_n^{\mu(a_n)}$. Clearly, for each permutation w' of w_μ we have $\Psi_V(w') = \mu(V)$, where Ψ_V is the Parikh mapping associated with V.

Conversely, with each string $x \in V^*$ we can associate the multiset $\mu_x : V \longrightarrow \mathbf{N}$ defined by $\mu_x(a) = |x|_a$, for each $a \in V$.

For a multiset μ we denote by $|\mu|$ its *weight*, defined by $|\mu| = \sum_{a \in V} \mu(a) = |w_\mu|$; for $U \subseteq V$, we also denote $|\mu|_U = \sum_{a \in U} \mu(a) = |w_\mu|_U$.

A (finite or infinite) set of multisets over an alphabet V is called a *macroset*. Thus, $V^{\#}$ is the *universal macroset* over V (it corresponds to the "universal language" V^* over V). A macroset $M \subseteq V^{\#}$, for $V = \{a_1, \ldots, a_n\}$ can be naturally represented by the set of vectors $\{(\mu(V)) \mid \mu \in M\}$.

For two multisets μ_1, μ_2 over the same V we define the *inclusion* $\mu_1 \subseteq \mu_2$, the *sum* $\mu_1 + \mu_2$, the *union* $\mu_1 \cup \mu_2$, the *intersection* $\mu_1 \cap \mu_2$, and, for the case $\mu_1 \subseteq \mu_2$ only, the *difference* $\mu_1 - \mu_2$ in the following ways:

$$\begin{aligned}
&\mu_1 \subseteq \mu_2 \text{ iff } \mu_1(a) \leq \mu_2(a), \text{ for all } a \in V;\\
&(\mu_1 + \mu_2)(a) = \mu_1(a) + \mu_2(a), \text{ for each } a \in V;\\
&(\mu_1 \cup \mu_2)(a) = \max(\mu_1(a), \mu_2(a)), \text{ for each } a \in V;\\
&(\mu_1 \cap \mu_2)(a) = \min(\mu_1(a), \mu_2(a)), \text{ for each } a \in V;\\
&(\mu_1 - \mu_2)(a) = \mu_1(a) - \mu_2(a), \text{ for each } a \in V.
\end{aligned}$$

A usual set $U \subseteq V$ is a multiset μ_U with $\mu_U(a) = 1$ iff $a \in U$. The empty multiset is denoted by $\emptyset$ and it corresponds to the empty string λ.

The correspondence among strings and multisets (hence among languages and macrosets) suggests considering the following macrosets generating devices.

A multiset grammar is a construct $G = (N, T, A, P)$, where N, T are disjoint alphabets, the *nonterminal* and the *terminal* one, respectively (we denote their union by V), $A \subseteq V^{\#}$ is a finite macroset over V (its elements are called *axioms*, and P is a finite set of *multiset rewriting rules* (in short, *rules*) of the form $\mu_1 \rightarrow \mu_2$, where μ_1, μ_2 are multisets over V and $|\mu_1|_N \geq 1$. (Taking advantage of the string representation of multisets, we will usually write strings instead of multisets in A and in rules.)

For two multisets α_1, α_2 over V, we write $\alpha_1 \Longrightarrow_r \alpha_2$ for some $r : \mu_1 \rightarrow \mu_2 \in P$ if $\mu_1 \subseteq \alpha_1$ and $\alpha_2 = (\alpha_1 - \mu_1) + \mu_2$. If r is understood, then we write $\Longrightarrow$ instead of $\Longrightarrow_r$. We denote by $\Longrightarrow^*$ the reflexive and transitive closure of the relation $\Longrightarrow$. The macroset *generated* by G is defined by

$$M(G) = \{\beta \in T^{\#} \mid \alpha \Longrightarrow^* \beta, \text{ for some } \alpha \in A\}.$$

A natural Chomsky-like classification of such grammars is the following one:

1. Grammars G as above are said to be *arbitrary.*
2. If $|\mu_1| \leq |\mu_2|$ for all rules $\mu_1 \rightarrow \mu_2$ in P, then G is said to be *monotone.*
3. If $|\mu_1| = 1$ for all rules $\mu_1 \rightarrow \mu_2$ in P, then G is said to be *context-free.*
4. If $|\mu_1| = 1$ and $|\mu_2|_N \leq 1$ for all rules $\mu_1 \rightarrow \mu_2$ in P, then G is said to be *linear.*
5. If $|\mu_1| = 1, |\mu_2| \leq 2$, and $|\mu_2|_N \leq 1$ for all rules $\mu_1 \rightarrow \mu_2$ in P, then G is said to be *regular.*
   We also consider the following subclass of linear grammars, which corresponds to no Chomsky class:
6. If G is a linear multiset grammar such that for each rule $\mu_1 \rightarrow \mu_2$ in P such that $\mu_2(a_i) > 0, \mu_2(a_j) > 0$ for some $1 \leq i < j \leq n$, then $\mu_2(a_k) > 0$ for all $i \leq k \leq j$, then G is said to be *local.* (We always increase the number of copies of objects which are adjacent in the ordering of V.)

We denote by *mARB, mMON, mCF, mLIN, mREG, mLOC* the families of macrosets generated by arbitrary, monotone, context-free, linear, regular, and local multiset grammars, respectively. By *FIN, REG, LIN, CF, CS, RE* we denote the families of finite, regular, linear, context-free, context-sensitive, and recursively enumerable languages, respectively. For a family F of languages we denote by PsF the family of Parikh sets of vectors associated with languages in F. The family of all semilinear languages is denoted by $SLin$.

We also consider here *matrix grammars*, with and without appearance checking, both for the string and the multiset case. The definition for multisets is a direct extension of the definition for strings. Always we consider only matrix grammars with context-free rules. We denote by MAT the family of languages generated by matrix grammars without λ rules and without appearance checking; if erasing rules are used, then the superscript λ is added, if appearance checking features are used, then the subscript ac is added. For the multiset case we add the letter m in front of these notations, thus obtaining the families $mMAT, mMAT^{\lambda}, mMAT_{ac}$, and $mMAT_{ac}^{\lambda}$, respectively.

3 Generative Power

We start by proving a normal form result which says that instead of a finite set of axioms, we may consider one axiom only, in the form of a single nonterminal symbol (like in Chomsky grammars).

Lemma 1. *For each multiset grammar $G = (N, T, A, P)$, there is an equivalent multiset grammar $G' = (N', T, A', P')$, where A' contains only one multiset S over $N' \cup T$, such that $|S| = 1$. Moreover, G' is of the same type as G.*

Proof. For arbitrary, monotone, and context-free grammars the assertion is obvious: take one more nonterminal symbol, S, add it to N and add all rules $S \to w$, for $w \in A$, to P. We obtain an equivalent grammar, which is monotone (context-free) if G was monotone (context-free).

For the linear case we proceed as follows. Assume that $A = \{w_1, \ldots, w_n\}$, with $w_i \in (N \cup T)^{\#}$. Let $N(w_i), T(w_i)$ be the multisets of nonterminals and of terminals, respectively, which appear in $w_i, 1 \leq i \leq n$, and denote $t = \max\{|N(w_i)| \mid 1 \leq i \leq n\}$. Then, consider

$$\begin{aligned}
N' &= \{[w] \in N^* \mid 1 \leq |w| \leq t\} \cup \{S\}, \\
A' &= \{S\}, \\
P' &= \{S \to [N(w_i)]T(w_i) \mid 1 \leq i \leq n\} \\
&\cup \{[w] \to [w']x \mid [w], [w'] \in N', w' = (w - X) + Y, \text{ for a rule} \\
&\quad\; X \to Yx \in P, X, Y \in N, x \in T^*\} \\
&\cup \{[X] \to x \mid \text{for } X \to x \in P, X \in N, x \in T^*\}.
\end{aligned}$$

The equality $M(G) = M(G')$ is obvious and also obvious is the fact that G' is linear, regular, local if G is linear, regular, local, respectively. □

Because most of the relationships between the above mentioned families of macrosets are easy to be established, we directly give a synthesis result (also including a relation which will be proved later):

Theorem 1. (Macroset Hierarchy Theorem) *The relations in the diagram from Figure 1 hold; the arrows denote inclusions of the lower families into the upper families; all these inclusions are proper, with the exception of the two arrows marked with a question mark, for which the properness is open.*

Proof. In view of Lemma 1, we can always assume that the set of axioms contains only one axiom, consisting of a single nonterminal.

The equalities $mREG = PsREG, mLIN = PsLIN$ and $mCF = PsCF$ are obvious; because $PsREG = PsLIN = PsCF = SLin$, we get the equality of all these families.

It is clear that $mLOC$ contains infinite macrosets.

Consider the macroset $M = \{(n, 1, n) \mid n \geq 1\}$, for $V = \{a_1, a_2, a_3\}$. It is easy to see that we cannot generate this macroset by a local multiset grammar (all vectors $(n, 1, n), n \geq 1$, belong to M, hence we have to increase the number of a_1 and a_3 occurrences in a synchronized way, but we cannot do this in a local way), but we have $M \in mLIN$. This shows that $mLOC \subset mLIN$ is a proper inclusion.

Given a monotone multiset grammar $G = (N, T, A, P)$, we can construct a matrix multiset grammar in the following way. For each $X \in N \cup T$ consider the new symbols X', X''. For each rule $X_1X_2 \dots X_k \to Y_1Y_2 \dots Y_r$ in P, with $k \leq r$ (we use the string representation of rules), consider the matrix

$$(X_1' \to Y_1'', \dots, X_{k-1}' \to Y_{k-1}'', X_k' \to Y_k'Y_{k+1}' \dots Y_r', Y_1'' \to Y_1', \dots, Y_{k-1}'' \to Y_{k-1}').$$

(The use of double primed symbols prevents the rewriting of an object introduced by a previous rule of the same matrix; this is allowed in matrix grammars but not in monotone grammars.) The matrix grammar with these matrices, with all primed symbols as nonterminals, and with the axioms obtained from the axioms of A by priming their objects generate the same macroset as G. The technical details are left to the reader. Thus, $mMON \subseteq mMAT$.

Now, the equality $mMAT = PsMAT$ is obvious. (A normal form as that in the Fact we have started with also holds for matrix grammars. See also [8].) From a string matrix grammar $G = (N, T, S, R)$ in the binary normal form we can immediately pass to a multiset matrix grammar G' (for $(X \to Y, A \to x) \in R$ consider the multiset rewriting rule $XA \to Yx$, etc) such that $M(G') = \Psi_T(L(G))$. Consequently, $mMAT \subseteq mMON$, and therefore $mMON = mMAT = PsMAT$.

Because MAT contains non-semilinear languages, the inclusion $mCF \subset mMON$ is proper.

The equality $mMAT_{ac} = PsMAT_{ac}$ is obvious. $PsMAT \subset PsMAT_{ac}$ is a proper inclusion, because the one-letter languages in MAT^{λ} are regular, while MAT_{ac} contains the language $\{a^{2^n} \mid n \geq 1\}$. Therefore, both the inclusions $mMON \subset mMAT_{ac}$ and $mMAT^{\lambda} \subset mMAT_{ac}^{\lambda}$ are proper.

The equalities $mARB = mMAT^{\lambda} = PsMAT^{\lambda}$ follow in the same way as $mMON = mMAT = PsMAT$.

The equality $mMAT_{ac}^{\lambda} = PsMAT_{ac}^{\lambda}$ is obvious, $PsRE = PsMAT_{ac}^{\lambda}$ follows from $RE = MAT_{ac}^{\lambda}$. Because there are one-letter languages in RE which are not in CS, the inclusion $PsCS \subset mMAT_{ac}^{\lambda}$ is proper.

The inclusion $PsMAT_{ac} \subseteq PsCS$ follows from $MAT_{ac} \subset CS$. □

Observe the unexpected relations $mMON = mMAT$, $mARB = mMAT^{\lambda}$, $mMON \subset mMAT^{\lambda} \cap mMAT_{ac}$, $mARB \subset mMAT_{ac}^{\lambda}$, $mMON \subset PsCS$, which correspond to relations from the language case which are different, unknown, or even opposite (this last case appears for $mMON \subset mMAT_{ac}$ versus $MAT_{ac} \subset CS$).

Quite interesting is also the fact that $mARB \subset PsRE$ is a strict inclusion. This means that arbitrary multiset rewriting rules are not sufficient in order to get a characterization of the power of Turing machines as multiset processing devices. Regulated rewriting of multisets (following the model of regulated rewriting in formal language theory, [9]) is thus necessary and, at least for matrix grammars with appearance checking, we get a characterization of $PsRE$.

Many such characterizations can be found in the distributed framework of P systems; the previous observation that $mARB \subset PsRE$ can also be seen as a further motivation of membrane computing.

$$\begin{tikzcd}
 & mMAT^{\lambda}_{ac} = PsRE = PsMAT^{\lambda}_{ac} & \\
 & & PsCS \arrow[ul] \\
mARB = mMAT^{\lambda} = PsMAT^{\lambda} \arrow[uur] & & mMAT_{ac} = PsMAT_{ac} \arrow[u, "?"'] \\
 & mMON = mMAT = mARB_{lb} = PsMAT \arrow[ul, "?"] \arrow[ur] & \\
 & mREG = mLIN = mCF = PsREG = PsLIN = PsCF = SLin \arrow[u] & \\
 & mLOC \arrow[u] & \\
 & mFIN = PsFIN \arrow[u] &
\end{tikzcd}$$

Figure 1: The macroset families hierarchy

In the diagram in Figure 1 there appears a family which was not defined yet, $mARB_{lb}$. This is the family of macrosets which can be generated by arbitrary multiset grammars with a *linearly bounded workspace*.

For $G = (N, T, A, P)$, let $\sigma : \alpha \Longrightarrow^* \beta$ be a derivation with $\alpha \in A$. The workspace of σ, denoted by $WS(\sigma)$, is the maximal weight of multisets appearing in σ. Then, for $\beta \in T^{\#}$ we put

$$WS_G(\beta) = \inf\{WS(\sigma) \mid \sigma : \alpha \Longrightarrow^* \beta, \alpha \in A\}.$$

We say that G has a linearly bounded workspace if there is a constant k such that for each non-empty $\beta \in M(G)$ we have $WS(\beta) \leq k|\beta|$.

The family of all macrosets generated by grammars with a linearly bounded workspace is denoted by $mARB_{lb}$.

The following result corresponds to the workspace theorem in formal language theory; the proof for multisets is simpler than in the case of languages.

Theorem 2. (Workspace Theorem) $mMON = mARB_{lb}$.

Proof. We have only to prove the inclusion $mARB_{lb} \subseteq mMON$.

Consider an arbitrary multiset grammar $G = (N, T, A, P)$ and a constant k such that $WS_G(\beta) \leq k|\beta|$ for all non-empty $\beta \in M(G)$. We construct the monotone multiset grammar $G' = (N', T, S, P')$ with

$$N' = \{\langle u\rangle \mid u \in (N \cup T \cup \{b\})^{\#}, |u| = k\} \cup \{S\},$$

where S and b are new symbols, and the following rules (we denote $V = N \cup T \cup \{b\}$):

1. $S \to \langle v_1\rangle \dots \langle v_q\rangle \langle v_{q+1}b^t\rangle S$,
   $S \to \langle v_1\rangle \dots \langle v_q\rangle \langle v_{q+1}b^t\rangle$
   for $v_1 \dots v_q v_{q+1} \in A, |v_i| = k, 1 \leq i \leq q, q \geq 0$, and $|v_{q+1}b^t| = k, t \geq 0$,

2. $S \to \langle b^k\rangle S$,

3. $\langle u\rangle\langle v\rangle \to \langle u'\rangle\langle v'\rangle$
   for all $u, v, u', v' \in V^{\#}, |u| = |v| = |u'| = |v'| = k, u + v = u' + v'$,

4. $\langle u_1\rangle \dots \langle u_r\rangle \to \langle v_1\rangle \dots \langle v_r\rangle$,
   for each $\langle u_i\rangle, \langle v_i\rangle \in N', 1 \leq i \leq r, r \geq 1$, such that $u_1 + u_2 + \dots + u_r \Longrightarrow v_1 + v_2 + \dots + v_r$ by a rule $u \to v \in P$,

5. $\langle u_1\rangle \dots \langle u_r\rangle \to \langle v_1\rangle \dots \langle v_r\rangle\langle v_{r+1}\rangle \dots \langle v_t\rangle$,
   for $\langle u_i\rangle, \langle v_j\rangle \in N', 1 \leq i \leq r, 1 \leq j \leq t, r \geq 1, t > r$, such that $u_1 + u_2 + \dots + u_r \Longrightarrow v_1 + v_2 + \dots + v_r + v_{r+1} + \dots + v_t$ by a rule $u \to v \in P$ and $|v_i|_b = 0$ for all $1 \leq i \leq t-1, v_t \in V^{\#}, |v_t| = k$,

6. $\langle u\rangle \to u$, for all $u \in (T \cup \{b\})^{\#}$ with $|u|_T \geq 1$.

The nonterminals of G' are multisets of weight k, maybe with multiple occurrences of the object b. By rules of type1 we introduce the axioms of G, splitted among nonterminals of G'; further nonterminals of the form $\langle b^k\rangle$ can be introduced, if needed in the derivation, by rules of type 2. By rules of type 3, the contents of multisets in the nonterminal symbols of G' can be freely interchanged. By rules of types 4 and 5 we can simulate the rules from P, by making use of the "free space" made available by the occurrences of b in the nonterminals of G'. When no nonterminal with respect to G is present in a nonterminal $\langle u\rangle$, then by rules of type 6 we introduce the terminal objects of u, providing that at least one exist; the occurrences of b are ignored. Thus, one can easily see that we get $M(G) = M(G')$. $\square$

The previous result is related to the *open problem* whether or not the inclusion $mMON \subseteq mARB$ is proper. If true, the equality $mARB = mARB_{lb}$ would be another result surprisingly different from that known from the case of languages.

4 An Algebraic Characterization

Another way to characterize the families *mREG, mLIN, mCF, PsREG, PsLIN*, and *PsCF* is to define macrosets as least fixed points (solutions) of corresponding systems of equations on the ω-complete semiring with vector addition as underlying operation (for the theory of semirings, see [11], [14]). This can be done in a quite general manner for other operations, giving rational (corresponding to regular), linear, and algebraic (corresponding to context-free) languages/macrosets. If the underlying operation is commutative all these families of languages/macrosets coincide and this is the case for vector addition on $\mathbf{N}^n$.

Let $\mathcal{M}$ be a monoid with binary operation $\circ$ and unit element $\mathbf{1}$, or with a binary operation $\circ : \mathcal{M} \times \mathcal{M} \to \mathcal{P}(\mathcal{M})$ with unit element $\mathbf{1}$, i.e., $\mathbf{1} \circ \alpha = \alpha \circ \mathbf{1} = \{\alpha\}$.

Extend $\circ$ to an associative binary operation $\circ : \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathcal{M}) \to \mathcal{P}(\mathcal{M})$, being distributive with union $\cup$ $(A \circ (B \cup C) = (A \circ B) \cup (A \circ C)$ and $(A \cup B) \circ C = (A \circ B) \cup (B \circ C))$, with unit element $\{\mathbf{1}\}$ $(\{\mathbf{1}\} \circ A = A \circ \{\mathbf{1}\} = A)$, and zero element $\emptyset$ $(\emptyset \circ A = A \circ \emptyset = \emptyset)$.

Then $\mathcal{S} = (\mathcal{P}(\mathcal{M}), \cup, \circ, \emptyset, \{\mathbf{1}\})$ is an ω-complete semiring, i.e,. if $A_i \subseteq A_{i+1}$ for $0 \leq i$ then $B \circ \bigcup_{i \geq 0} A_i = \bigcup_{i \geq 0} (B \circ A_i)$ and $(\bigcup_{i \geq 0} A_i) \circ B = \bigcup_{i \geq 0} (A_i \circ B)$.

Define also $A^{(0)} = \{\mathbf{1}\}$, $A^{(1)} = A, A^{(k+1)} = A \circ A^{(k)}$, $A^{\circ} = \bigcup_{k \geq 0} A^{(k)}$.

Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ be a set of variables such that $\mathcal{X} \cap \mathcal{M} = \emptyset$.

A *monomial* over $\mathcal{S}$ with variables in $\mathcal{X}$ is a finite string of the form: $A_1 \circ A_2 \circ \ldots \circ A_k$, where $A_i \in \mathcal{X}$ or $A_i \subseteq \mathcal{M}, |A_i| < \infty, i = 1, \ldots, k$. Without loss of generality, $A_i = \{\alpha_i\}$ with $\alpha_i \in \mathcal{M}$ suffices. The α_{ij} (or $\{\alpha_{ij}\}$) will be called *constants*. A *polynomial* $p(\underline{X})$ over $\mathcal{S}$ is a finite union of monomials where $\underline{X} = (X_1, \cdots, X_n)$.

A *system of equations* over $\mathcal{S}$ is a finite set of equations

$$\mathcal{E} := \{X_i = p_i(\underline{X}) \mid i = 1, \ldots, n\},$$

where $p_i(\underline{X})$ are polynomials. This will also be denoted by $\underline{X} = \underline{p}(\underline{X})$.

The *solution* of $\mathcal{E}$ is a n-tuple $\underline{L} = (L_1, \ldots, L_n)$ of sets over $\mathcal{M}$, with $L_i = p_i(L_1, \ldots, L_n)$ and the n-tuple is minimal with this property, i.e., if $\underline{L'} = (L'_1, \ldots, L'_n)$ is another n-tuple satisfying $\mathcal{E}$, then $\underline{L} \leq \underline{L'}$ (where the order is defined componentwise with respect to inclusion, i.e., $\underline{A} = (A_1, \cdots, A_n) \leq (B_1, \cdots, B_n) = \underline{B} \Leftrightarrow \forall_{i=1}^n : A_i \subseteq B_i$).

From the theory of semirings it follows that any system of equations over $\mathcal{S}$ has a unique solution, and this is the least fixed point starting with

$$\underline{X}^{(0)} = (X_1^{(0)}, \cdots, X_n^{(0)}) = (\emptyset, \cdots, \emptyset) = \underline{\emptyset}, \text{ and } \underline{X}^{t+1} = \underline{p}(\underline{X}^{(t)}).$$

Then the following fact holds: $\underline{X}^{(t)} \leq \underline{X}^{(t+1)}$ for $0 \leq t$.

This is seen by induction and the property of the polynomial with respect to inclusion, as $\underline{\emptyset} \leq \underline{X}^{(1)}$ and $\underline{X}^{(t+1)} = \underline{p}(\underline{X}^{(t)}) \leq \underline{p}(\underline{X}^{(t+1)}) = \underline{X}^{(t+2)}$.

A general system of equations is called *algebraic*; it is *linear* if all monomials are of the form $A \circ X \circ B$ or A, and *rational* if they are of the form $X \circ A$ or A, with $A \subseteq M$ and $B \subseteq M$. Corresponding families of macrosets (solutions of such systems of equations) are denoted by $\underline{ALG(\circ)}$, $\underline{LIN(\circ)}$, and $\underline{RAT(\circ)}$. In the case $\circ$ is commutative then all families are identical: $\underline{ALG(\circ)} = \underline{LIN(\circ)} = \underline{RAT(\circ)}$.

Interpreting the variables of the system of equations as non-terminals also rational (regular), linear and algebraic (context-free) grammars can be defined. They generate the same languages as those defined by least fixed points.

Since the operation of vector addition is commutative it follows that rational, linear and algebraic languages coincide. In this way, we have again obtained the corresponding equalities from Theorem 1.

We close this section by pointing out one more associative operation on multisets. Consider

$$M = \bigcup_{k=0}^{\infty} \mathbf{N}^{|\Sigma|^k} - \bigcup_{k=1}^{\infty} \{0\}^{|\Sigma|^k}$$

($\mathbf{N}^0 = \{\mathbf{1}\}$ where $\mathbf{1}$ is considered as a unit element).

Interprete $x \in M$ as a multiset representing the multiplicities of words of length k in lexicographical order. An operation $\odot : M \times M \to M$ is defined in the following way:

$$x \odot y = \langle \xi \cdot \eta \mid \xi \in x, \eta \in y \rangle, \ \mathbf{1} \odot x = x \odot \mathbf{1} = x,$$

respecting all multiplicities, and where $\cdot$ is the catenation.

Example:

$$\langle a, a, b \rangle \odot \langle aa, ba \rangle = \langle aaa, aaa, aba, aba, baa, bba \rangle,$$

or, in other notation, $(2,1) \odot (1,0,1,0) = (2,0,2,0,1,0,1,0)$.

$\odot$ is an associative operation since

$$(x \odot y) \odot z = \langle \xi \cdot \eta \cdot \zeta \mid \xi \in x, \eta \in y, \zeta \in z \rangle = x \odot (y \odot z).$$

Extending $\odot$ to $\mathcal{P}(M)$ gives a monoid $\mathcal{M} = (\mathcal{P}(M), \odot, \mathbf{1})$ and an ω-complete semiring $\mathcal{S} = (\mathcal{P}(M), \cup, \odot, \emptyset, \{\mathbf{1}\})$.

5 Conclusion

Of course, many other problems remain to be investigated; the whole program of formal language theory can/must be repeated for macrosets. For instance, we can define pure, distributed (like in grammar system area), or parallel (like in Lindenmayer and in P system area) multiset grammars; also other regulating mechanisms than the matrix one can be considered. We leave to the interested reader the task to make the further step toward Formal Macroset Theory.

References

[1] J. P. Banâtre, A. Coutant, D. Le Metayer, A parallel machine for multiset transformation and its programming style, *Future Generation Computer Systems*, 4 (1988), 133–144.

[2] J. P. Banâtre, D. Le Metayer, The Gamma model and its discipline of programming, *Science of Computer Programming*, 15 (1990), 55–77.

[3] J. P. Banâtre, D. Le Metayer, Gamma and the chemical reaction model: ten years after, *Coordination Programming: Mechanisms, Models and Semantics*, Imperial College Press, 1996.

[4] H. P. Barendregt, *The Lambda Calculus; Its Syntax and Semantics*, North-Holland, Amsterdam, 1984.

[5] T. Basten, Parsing partially ordered multisets, *Intern. J. Found. Computer Sci.*, 8, 4 (1997), 379–407.

[6] G. Berry, G. Boudol, The chemical abstract machine, *Theoretical Computer Sci.*, 96 (1992), 217–248.

[7] A. Colmerauer, Equations and inequalities on finite and infinite trees, in *Proc. of Second Intern. Conf. on Fifth Generation Computer Systems*, Tokyo, 1984, 85–99.

[8] S. Crespi-Reghizzi, D. Mandrioli, Commutative Grammars, *Calcolo*, vol. XIII, fasc. II (1976), 173-189.

[9] J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.

[10] J. L. Gischer, The equational theory of pomsets, *Theoretical Computer Sci.*, 61, 2-3 (1988), 199–224.

[11] J. S. Golan, *The Theory of Semirings with Application in Mathematics and Theoretical Computer Science*, Longman Scientific and Technical, 1992.

[12] S. Gorn, Explicit definitions and linguistic dominoes, in J. Hart, S. Takasu, eds., *Systems and Computer Science*, Univ. of Toronto Press, Toronto, 1967, 77–115.

[13] J. Kortelainen, Properties of trios and AFLs with bounded or commutative generators, *Mathematics, University of Oulu*, No. 53, 1980.

[14] W. Kuich, A. Salomaa, *Semirings, Automata, Languages*, EATCS Monographs on Theoretical Computer Science 5, Springer-Verlag, Berlin,

[15] M. Latteux, Cônes rationnels commutativement clos, *R.A.I.R.O.*, 11, 1 (1977), 29–51.

[16] M. Latteux, Langages commutatifs, Publications de Laboratoire de Calcul de l'Université des Sciences et Techniques de Lille, Mai 1978.

[17] M. Latteux, Cônes rationnels commutatifs, *JCSS*, 18 (1979), 307–333.

[18] S. Miyamoto, Fuzzy multisets and application to rough approximation of fuzzy sets, *Techn. Rep. Inst. of Inform. Sciences and Electronics*, Univ. of Tsukuba, 1SE-TR-96-136, June 1996, 1–10.

[19] S. Miyamoto, Basic operations on fuzzy multisets, *J. of Japan Soc. of Fuzzy Theory and Systems*, 8, 4 (1996).

[20] S. Miyamoto, Fuzzy multisets with infinite collections of memberships, *Proc. 7th Intern. Fuzzy Systems Ass. World. Congress* (IFSA'97), June 1997, Prague, vol. I, 61–66.

[21] B. Li, Fuzzy bags and applications, *Fuzzy Sets and Systems*, 34 (1990), 61–71.

[22] Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61 (2000), in press, and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).

[23] Gh. Păun, Computing with membranes. An introduction, *Bulletin of the EATCS*, 67 (Febr. 1999), 139–152.

[24] Gh. Păun, Computing with membranes (P systems): Twenty six research topics, this volume and *Auckland University, CDMTCS Report* No 119, 2000 (www.cs.auckland.ac.nz/CDMTCS).

[25] Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Heidelberg, 1998.

[26] G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, 3 volumes, Springer-Verlag, Berlin, 1997.

[27] Y. Suzuki, H. Tanaka, Symbolic chemical system based on abstract rewriting system and its behavior pattern, *Artificial Life Robotics*, 1 (1997), 211–219.

[28] Y. Suzuki, H. Tanaka, Chemical evolution among artificial proto-cells, *Proc. of Artificial Life VIII Conf.*, MIT Press, 2000

[29] A. Syropoulos, A note on multi-sets, basic pairs and the chemical abstract machine, manuscript, 2000.

[30] A. Syropoulos, Fuzzy sets and fuzzy multisets as Chu spaces, manuscript, 2000.

[31] A. Syropoulos, *Multisets and Chu Spaces*, PhD Thesis, Univ. of Xanti, Greece, in preparation.

[32] R. R. Yager, On the theory of bags, *Intern. J. General Systems*, 13 (1986), 23–37.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 159 - 175.

Membrane Computing in Prolog

Mihaela Maliţa
Faculty of Mathematics, University of Bucharest
str.Academiei 14, 70109 Bucharest, ROMANIA
Email: malita@rnc.ro

Abstract

This paper presents a simulation environment written in Prolog for the new paradigm of computation proposed by G. Păun in "Computing with Membranes" [1]. We present from [1] the concept and our Prolog predicates dealing with the main operations on multisets, on membrane structures and on super cell systems. The program was tested on the examples from [1]. The simulator is conceived to be useful for developing applications, which might test the power of this new computing paradigm.

1 Introduction

In November 1998 Gheorghe Păun at the Center for Computer Science, Turku, Finland proposed a new paradigm of computation: Membrane Computation [1].

The Membrane Computation seems to be an interesting and original approach to computing inspired from biochemestry.

Meantime this new paradigm captured the attention of many other researchers (J.Dassow [2], G.Rozenberg [3], A.Salomaa [3]).

The main new concepts defined are: the membrane, membrane structure and super cell system [1].

Inspired from biochemestry this model is based on the concept of membrane represented in his formalism by the mathematical concept of multiset.

A membrane structure is a set of labeled multisets with certain restrictions. On the membrane structure Păun adds a set of transforming rules and he obtains thus the super cell system.

The calculus is the evolution of a super cell system from its initial state according to the given rules. Applying rules and working with symbol manipulation is appropriate for programming in Prolog. Although our Prolog [5] is not concieved for parallel programming we choosed this approach taking into account its other facilities.

When we define a new concept the question arises: "which is the most appropriate representation for it?". The membrane computation paradigm starts from the concept of multiset. In our paper we followed in detailed each step in the presentation of the model in order to make the code transparent.

Let us take the intuitive definition of the multiset: *an "extended" set where we admit that the elements could occure multiple times.* In our paper we shall always give an example of calling a predicate (the `?-` is the Prolog prompt) and then the answer of the Prolog interpreter. Let us show as an example how we write the Prolog predicate `multiset(List)` which tests if a list is representing a multiset.

Example 1 *The multiset* $\{a,a,b,c,c,c\}$ *is in Prolog:* `[a(2),b(1),c(3)].` ⋄

multiset(Ms). Tests if a list Ms is a Multiset.

```
?- multiset([a(1),b(2),c(4)]).
yes
```

Then we shall write the clauses defining the predicate.

```
object(_(N)):-integer(N).
multiset([]).
multiset([H|T]):-object(H),multiset(T).
```

In our paper all the predicates follow the same previous pattern. Some definitons are very simple as those in section 2, dealing with the well known concept of multiset. We decided to present them also in detail as a Prolog exercise: how to use the list representation for multisets.

The most difficult thing was the simulation of paralelism of the membrane computing model in Prolog.

2 Multisets in Prolog. Basic Operations

In the following, we present multisets in the list representation for Prolog.

Representation of a multiset. A multiset is a set where we admit multiple occurrences of its elements now named objects. In Prolog, we shall represent a multiset as a list of objects together with their multiplicity.

multiplicity(Object,Muliset). The multiplicity of an object from a multiset is the number of occurrences of the object.

```
?-multiplicity(a,[a(2),b(3),c(8)],M).
M=2
```

```
multiplicity(H,Ms,R):-on(H(R),Ms),!.
```

Usually objects that have multiplicity 0 are of no use and we can delete them from the multiset. The following predicate describes this property:

```
mult_zero(_(0)).
```

Support of a multiset is the set (ignoring the multiplicity).

support(Ms,R). From the multiset Ms we select the objects without their multiplicity. The resulting set is R.

```
?- support([a(2),b(3),c(8)],R).
R=[a,b,c]
```

```
support([],[]).
support([H(_)|T],[H|R]):- support(T,R).
```

Inclusion of multisets. X is included in Y if all the objects from X are also in Y and their multiplicity in X is smaller or equal than their multiplicity in Y.

include(X,Y). Returns yes if X $\subset$ Y else no.

```
    ?- include([a(2),b(3)],[a(3),b(4),c(7)]).
    yes
```

```
include([],_).
include([H(R1)|T],L):- on(H(R2),L),R1=<R2,include(T,L).
```

In the following, we use the classical *concat(L1,L2,R)* for concatenating two lists L1 and L2:

```
concat([],Y,Y).
concat([H|T],Y,[H|R]):- concat(T,Y,R).
```

Union of multisets. The union of two multisets is a multiset, which contains all the objects from X and Y, and their multiplicity is the sum of their multiplicity in X and Y.

union(X,Y,R). X $\cup$ Y = R

```
    ?- union([a(2),b(3)],[a(4),c(8)],R).
    R=[a(6),b(3),c(8)]
```

```
union([],Y,Y).
union([H(R1)|T],Y,[H(R)|S]):- concat(Y1,[H(R2)|Y2],Y),
            R is R1+ R2,concat(Y1,Y2,Ynew),union(T,Ynew,S),!.
union([H(R1)|T],Y,[H(R1)|S]):- union(T,Y,S).
```

Difference of multisets. The difference is defined only if the multisets are included one in another. The difference is a multiset with the objects from M1 and their multiplicity is the difference between the multiplicity in M1 and the corresponding multiplicity in M2.

difference(M1,M2,Dif). M1 - M2 = Dif

```
    ?- difference([a(2),b(3),c(3)],[a(2),b(1)],R).
    R=[b(2),c(3)].
    ?- difference([a(2),b(3)],[a(2),b(3)],R).
    R=[].
    ?- difference([a(2),b(3)],[b(5)],R).
    no
```

```
difference(Y,[],Y):- !.
difference(Y,[H(R2)|T],RR):- concat(Y1,[H(R1)|Y2],Y),R1 >= R2,
            R is R1-R2,concat(Y1,Y2,Ynew),difference(Ynew,T,S),
            (R = 0, RR=S,!; R > 0, RR=[H(R)|S]).
```

3 The Membrane Structure in Prolog

A membrane is an labeled multiset.

A membrane structure is a structure composed by labeled paranthesis. Example of a membrane structure: [[ ] [[ ]]. If we fill this structure with elements from an alphabet. Let us consider A=a,b,c) the following is an example of a super cell: [a,a,[b,a],[c,[ ],a].

A filled membrane with objects from an alphabet is actually a multiset.

In the super cell membranes structures cannot have any intersection. Membranes are disjoint or included one in another. The biological interpretation is that membranes don't intersect.

Example 2 *Let's take Păun's first example from [1] of a super cell.*

```
Ms= [a,a,c,[a,[c,d]],[]]
```

The representation that we work with is:

```
Ms=[a(2),c(1),[a(1),[c(1),d(1)]],[]].
```

◇

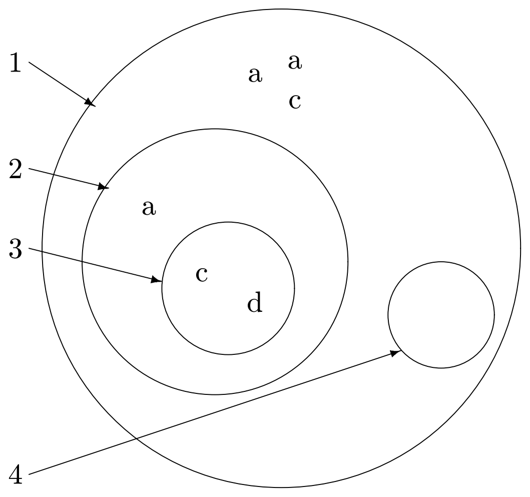


Fig. The diagram of Ms.

The membranes are labeled with numbers, so we could refer them in the super cell. Let's put numbers in the front of each sublist. Each sublist represents a membrane. There are four membranes and the membrane labeled 4 is empty.

```
Ms=[1,a(2),c(1),[2,a(1),[3,c(1),d(1)]],[4]]
```

Let's try to work with this representation. Advantages:

- the list structure respects the initial topology of the membrane

- the membranes are labeled with numbers (different!).

membrane(Nr,Ms,Mnr). In the list Mnr we obtain the content of membrane with number Nr from the super cell Ms.

```
        ?-membrane(2,[1,a(2),[2,a(3),b(4),[3,f(1),j(7)]]],[4]],R).
        R=[a(3),b(4)]
        ?-membrane(4,[1,a(2),[2,a(3),b(4),[3,f(1),j(7)]]],[4]],R).
        R=[]
        ?-membrane(5,[1,a(2),[2,a(3),b(4),[3,f(1),j(7)]]],[4]],R).
        no
```

```
object(_(_)):-!.
object(X):- atomic(X),not X=[].
membrane(K,[K|T],R):- select_p(object,T,R),!.
membrane(K,[H|T],R):- not atomic(H),membrane(K,H,R).
membrane(K,[H|T],R):- membrane(K,T,R).
```

dissolve(K,Ms,R). Dissolve the membrane K of the super cell Ms. The resulting super cell is R. To dissolve a membrane all its content is poured into the membrane in which it is contained (included). Moreover, the membrane disappears. In a list representation that means the brackets of the membrane are removed. If nothing could be dissolved then the function returns the initial super cell

```
        ?-dissolve(2,[1,a(2),c(1),[2,a(1),[3,c(1),d(1)]],[]],R).
        R=[1,a(2),c(1),a(1),[3,c(1),d(1)],[4]]
        ?-dissolve(3,[1,a(1),[2,c(2),[3,c(2)]]],R).
        R=[1,a(1),[2,c(4)]]
        ?-dissolve(4,[1,a(1),[2,b(2),[3,c(2)]]],R).
        R=[1,a(1),[2,b(2),[3,c(2)]]]
```

```
dissolve(K,[],[]):-!.
dissolve(K,M,R):- concat(M1,M2,M),concat([[K|T]],Ig,M2),
concat(M1,T,R1),concat(R1,Ig,R),!.
dissolve(K,[H|T],[H|Tr]):- atomic(H),dissolve(K,T,Tr),!.
dissolve(K,[H|T],[Hr|Tr]):- dissolve(K,H,Hr),dissolve(K,T,Tr).
```

where_is(K,Ms,N). We want to know the number N of the membrane that is in the super cell Ms and includes membrane K.

```
        ?- where_is(3,[1,a(1),[2,b(3),[3]],[4]],I).
        R=2
        ?- where_is(5,[1,a(1),[2,b(3),[3]],[4]],I).
        no
where_is(K,[],_):- fail.
where_is(K,[H|T],H):- number(H),on([K|_],T),!.
where_is(K,[H|T],R):- atom(H),where_is(K,T,R).
where_is(K,[H|T],R):- where_is(K,H,R),!  ; where_is(K,T,R),!.
```

transf(K,M,Ms,R). Transforms the super cell Ms by replacing exclusively the content of membrane K with a new multiset M. The resulting super cell is R.

```
        ?-transf(2,[b(3)],[1,a(2),[2,a(1),[3,c(1),d(1)]],[4]],R)
        R= [1,a(2),[2,b(3),[3,c(1),d(1)]],[4]]
        ?-transf(5,[b(3)],[1,a(2),[2,[3,c(1),d(1)]],[4]],R)
        no
```

```
transf(K,M,[],[]):-!.
transf(K,M,[K|T],[K|R]):-delete_p(object,T,Tr),concat(M,Tr,R),!.
transf(K,M,[H|T],[H|R]):-(object(H);number(H)),transf(K,M,T,R),!.
transf(K,M,[H|T],[Hr|Tr]):-transf(K,M,H,Hr),transf(K,M,T,Tr).
```

4 The Super Cell System (P -system)

The super cell system or P-system is a super cell together with a set of rules of transformation [1].

The super cell evolves according to the rules. The rules are applied in parallel for each membrane. In [1] Gh. Păun defines some types of rules. We implemented them in Prolog in section 3.2. In the following, we present some necessary Prolog predicates used in the program but not specially characteristic with the membrane calculus.

4.1 Necessary Predicates for our Program

subst_all(V,N,Old_list,New_list). Substitutes on all levels in Old_list the old object V by the new object N.

```
        ?- subst_all(x,y,[1,a(1,x),[2,b(1,x]]).
        R= [1,a(1,y),[2,b(1,y]]
```

```
subst_all(V,N,X,X):- atomic(X),!.
subst_all(V,N,O(X,Y),O(X,Y)):- not Y=V,!.
subst_all(V,N,[O(F,V)|T],[O(F,N)|R]):- subst_all(V,N,T,R),!.
subst_all(V,N,[H|T],[R1|R2]):- subst_all(V,N,H,R1),
                     subst_all(V,N,T,R2).
```

select_p(P,List,R). Selects all the elements from List (first level) which have a certain property P. The resulting list is R.

```
        ?- select_p(object,[1,a(1),c(4),[2,b(3)]],I).
        I=[a(1),c(4)]
```

```
select_p(P,[],[]):- !.
select_p(P,[H|T],[H|R]):- P(H),select_p(P,T,R).
select_p(P,[H|T],R):- select_p(P,T,R).
```

delete_p(P,List,R). Deletes all objects from the first level of List that have the property P. The resulting list is R.

```
        ?- delete_p(integer,[2,a,3,b],R).
        R=[2,3]
        ?- delete_p(mult_zero,[a(2),d(0),b(0),c(7)],R).
        R=[a(2),c(7)]

delete_p(P,[],[]).
delete_p(P,[X|T],R):- P(X),delete_p(P,T,R),!.
delete_p(P,[X|T],[X|R]):- delete_p(P,T,R).
```

4.2 Basic Operations in a Super Cell

In our program objects from a multiset look like: `a(1,x)` or `c(1,y)`. We shall explain later why we introduced the markers x and y for each object. This small modification obliged us to rewrite the main operations on multisets and membranes, presented in section 2 and section 3.

union(M1,M2,R). R is the union of membranes M1 and M2. It follows the same pattern as the union for multisets.

```
        ?- union([a(2,x),b(3,x)],[a(1,x),b(1,y)],R).
        R=[a(3,x),b(3,x)],b(1,y)]

union([],Y,Y).
union([H(R1,K1)|T],Y,[H(R,K1)|S]):- concat(Y1,[H(R2,K1)|Y2],Y),
            R is R1+ R2,concat(Y1,Y2,Ynew),union(T,Ynew,S),!.
union([H(R1,K)|T],Y,[H(R1,K)|S]):- union(T,Y,S).
```

difference(M1,M2,D). D is the difference of membranes. It follows the same pattern as for multisets section 2. If the multisets (membranes) are not included one in another the difference is not possible (fails).

```
        ?- difference([a(2,x),b(3,x),c(3,x)],[a(2,x),b(1,x)],R).
        R=[b(2,x),c(3,x)].
        ?- difference([a(2,x),b(3,x)],[a(2,x),b(3,x)],R).
        R=[].
        ?- difference([a(2,x),b(3,x)],[b(2,y)],R).
        no

difference(Y,[],Y):- !.
difference(Y,[H(R2,Mark)|T],RR):-
            concat(Y1,[H(R1,Mark)|Y2],Y), R1 >= R2,
            R is R1-R2,concat(Y1,Y2,Ynew),difference(Ynew,T,S),
            (R = 0, RR=S,!; R > 0, RR=[H(R,Mark)|S]).
```

membrane(K,Ms,Mk). Mk will be the content of membrane K from the membrane structure Ms (exclusively the content of membrane K).

```
        ?-membrane(2,[1,a(2,x),[2,b(4,x),[3,f(1,x)]]],[4]],R).
        R=[b(4,x)]

membrane(K,[K|T],R):- select_p(object,T,R),!.
membrane(K,[H|T],R):- not atomic(H),membrane(K,H,R).
membrane(K,[H|T],R):- membrane(K,T,R).
```

dissolve(K,Ms,R). Dissolve the membrane K in the membrane structure Ms. The resulting membrane structure is R.

```
        ?-dissolve(3,[1,a(1,x),[2,c(2,x),[3,b(1,x)]]],R).
        R=[1,a(1,x),[2,c(2,x),b(1,x)]]
```

```
dissolve(K,[],[]):- !.
dissolve(K,M,R):- concat(M1,M2,M),concat([[K|T]],Ig,M2),
                      concat(M1,T,R1),concat(R1,Ig,R),!.
dissolve(K,[H|T],[H|Tr]):- object(H),dissolve(K,T,Tr),!.
dissolve(K,[H|T],[Hr|Tr]):- dissolve(K,H,Hr),dissolve(K,T,Tr).
```

where_is(K,Ms,X). We want to know the number X of the membrane that includes K in the membrane structure Ms.

```
        ?- where_is(4,[1,a(1,x),[2,b(3,x),[3]],[4]],I).
        R=1
```

```
where_is(K,[],_):- fail.
where_is(K,[H|T],H):- number(H),on([K|_],T),!.
where_is(K,[H|T],R):- atom(H),where_is(K,T,R).
where_is(K,[H|T],R):- where_is(K,H,R),!  ; where_is(K,T,R),!.
```

in_order(Ms,R). Ms is a list (representing a super cell) where different objects may appear several times with different multiplicities. We want to put in order M that is each object must appear only once with the sum of all occurrences in a resulting list R, a correct written membrane structure .

```
        ?-in_order([1,a(1,x),c(1,x),a(1,x),[2,b(2,x),b(2,x)]],R).
        R=[1,a(2,x),c(1,x),[2,b(4,x)]]
```

```
in_order([],[]):-!.
in_order([N|T1],[N|T2]):- integer(N),in_order(T1,T2),!.
in_order([O(F,Y)|T1],[O(F,Y)|T2]):- not on(O(_,_),T1),in_order(T1,T2),!.
in_order([O(F,Y)|T1],R):- concat(X,[O(Fnew,Y)|Rest],T1),
              Fr is Fnew + F,concat(X,[O(Fr,Y)|Rest],T),in_order(T,R).
in_order([H|T],[Hr|Tr]):- in_order(H,Hr),in_order(T,Tr),!.
```

transf(K,M,Ms,R). This predicate transforms the membrane structure Ms by replacing the old membrane K with a new membrane M resulting the membrane structure R.

```
        ?-transf(2,[b(3,y)],
             [1,a(5,x),[2,a(1,x),c(1,x),[3,c(1,x),d(1,x)]],[4]],R)
        R= [1,a(5,x),[2,b(3,y),[3,c(1,x),d(1,x)]],[4]]
```

```
object(_(_,_)):-!.
object(X):- atomic(X),not X=[].
transf(K,M,[],[]).
transf(K,M,[K|T],[K|R]):-delete_p(object,T,Tr),concat(M,Tr,R),!.
transf(K,M,[H|T],[H|R]):-(object(H); number(H)),transf(K,M,T,R).
transf(K,M,[H|T],[Hr|Tr]):-transf(K,M,H,Hr),transf(K,M,T,Tr).
```

4.3 The Rules of a Super Cell System

A membrane structure that has rules concerning the transformation of its objects is called Super Cell System [1]. The rules could be defined, of course, in different ways. Here we take in consideration only the rules that appear in [1].

Let us start with an example from the following membrane structure:

```
Ms=[1,a(2,x),c(1,x),[2,a(1,x),[3,c(1,x),d(1,x)]],[4]]
```

For membrane 1, the following rule is proposed [1]:

```
(1) c → [in(4),a]
```

This means the number of the rule is 1 and we can apply it only for membrane 1. If membrane 1 has a c inside, then the c is moved and an object a appears in membrane 4. We write this rule in our data file as:

```
rule(1,1,[c(1,x)],[in(4),a(1,y)]).
```

The first 1 is the number of the membrane. The second 1 is the label of the rule. We label each rule by a number. The left side of the rule is the list of objects that are moved from membrane 1: [c(1,x)]. The right side of the rule says to put the objects from the list which begins with in(4) in membrane 4: [in(4),a(1,y)].

The Super Cell System is a parallel machine. The rules are applied simultaneously. In this version our solution to simulate the parallelism is to mark the new objects that appear in one clock with y. For rule [1,1] we write a(1,y) to make a difference from the objects marked with x which are not processed yet.

All the objects that appear in the right part of a rule are marked with y. This gives the possibility to make the difference between the new object and the old one in order not to apply at the same clock two different rules on the same object. We could also have a rule like that

```
(1) c,c,b → [in(4),a]
```

Our representation is:

```
rule(1,2,[c(2,x),b(1,x)],[in(4),a(1,y)]).
```

That means if membrane 1 contains two *c*'s and one *b* we take them and in membrane 4 we put an *a*.

We could also have a rule for the same membrane:

```
(4) b → a
```

In our representation:

```
rule(4,1,[b(1,x)],[a(1,y)]).
```

This means in membrane 4 we could change *b* with *a*. Actually all the *b*'s are transformed in *a*'s. Another principle is:

If a rule works for a membrane we apply the rule until it works no more.

We could also dissolve a membrane. Let's consider the rule:

```
(2) aac → dissolve
```

If membrane 2 has $a(2)$ and one c inside then delete the membrane. The content of the membrane is poured into the upper membrane.

Our representation:

```
rule(2,2,[a(2,x),c(1,x)],dissolve).
```

We could also throw out an object. This means the object is moved in the upper membrane (the "immediate" membrane which contains membrane 4):

```
     (4) c → [out,d].
rule(4,1,[c(1,x)],[out,d(1,y)]).
```

Let's take now each rule and see how it works following the Prolog program. All the rules are applied with the predicate

```
apply(rule(MembraneNr,RuleNr,Multiset,List), Ms,R).
```

R is the resulting membrane structure after applying the rule(..) on the membrane structure Ms.

rule(K,RuleNr,Mset,dissolve). If there is in the membrane structure a membrane K then dissolve membrane K, after we take Mset from it.

Example 3

```
rule(2,2,[a(1,x),c(1,x)],dissolve).
rule(3,1,[a(1,x)],dissolve).
        ?- apply(rule(2,1,[a(1,x)],dissolve),
                        [1,b(1,x),[2,a(2,x),[3]],[4]],R).
        R=[1,b(1,x),a(1,y),[3],[4]]

apply(rule(K,_,Mset,dissolve),Ms,R):- membrane(K,Ms,Mk),
              difference(Mk,Mset,New),modify_x(New,Newy),
              transf(K,Newy,Ms,Ms1),dissolve(K,Ms1,R),!,
              retractall(rule(K,_,_,_)).
```

◇

rule(K,RuleNr,Mset,[dissolve,ob(N,y),...]). If there is a multiset Mset in membrane K then dissolve membrane K, after you take Mset from it and put also all the objects Ob(N,y) from the list inside the membrane upper than K.

Example 4
```
rule(2,2,[c(1,x),a(1,x)],[dissolve,d(1,y)]).
        ?- apply(rule(2,_,[a(1,x)],[dissolve,b(1,y)]),
             [1,b(1,x),[2,a(2,x),[3]],[4]],R).
        R=[1,b(1,x),a(2,y),[3],[4]]

apply(rule(K,_,Mset,[dissolve|List]),Ms,R):- membrane(K,Ms,Mk),
             difference(Mk,Mset,New),union(New,List,U1),
             transf(K,U1,Ms,Ms1),dissolve(K,Ms1,R),!,
             retractall(rule(K,_,_,_)).
```
◇

rule(K,RuleNr,Mset,out). If membrane K has inside the multiset Mset then throw Mset out. That means that two membranes modify their content: membrane K and the membrane that includes K. The topology remains the same only the content of the membranes changed.

Example 5
```
rule(4,1,[c(1,x),d(1,x)],out).
        ?-apply(rule(2,[a(1,x)],out),
                        [1,a(3,x),c(1,x),[2,a(2,x),[3,d(1,x)]]],R).
        R=[1,a(4,x),c(1,x),[2,a(1,x),[3,d(1,x)]]]

apply(rule(K,_,Mset,out),Ms,R):- membrane(K,Ms,Mk),
             difference(Mk,Mset,D1),where_is(K,Ms,X),
             membrane(X,Ms,Mx),modify_x(Mset,Mset2),
             union(Mx,Mset2,Newx),transf(X,Newx,Ms,Ms1),
             transf(K,D1,Ms1,R),!.
```
◇

rule(K,RuleNr,Mset,[out|List]). Two membranes modify their content. From membrane K, we took Mset and in the membrane that includes K, we add all the objects from List. Applying the rule on the membrane structure Ms, we obtain a new membrane structure R.

Example 6
```
rule(4,1,[c(1,x)],[out,d(1,y)]).
        ?-apply(rule(2,[a(1,x)],[out,b(1,y)]),
                      [1,a(3,x),[2,a(2,x),[3,d(1,x)]]],R).
        R=[1,a(3,x),b(1,y),[2,a(1,x),[3,d(1,x)]]]

apply(rule(K,_,Mset,[out|List]),Ms,R):-membrane(K,Ms,Mk),
             difference(Mk,Mset,D1),where_is(K,Ms,X),
             membrane(X,Ms,Mx),union(Mx,List,Newx),
             transf(X,Newx,Ms,Ms1),transf(K,D1,Ms1,R),!.
```
◇

rule(K,RuleNr,Mset,[in(N)|MsetNew]). If membrane K contains the multiset Mset then we take Mset from it and put the multiset MsetNew in membrane N.

Example 7
```
rule(1,1,[c(1,x)],[in(4),a(1,y),b(2,y)]).
       ?-apply(rule(2,1,[a(1,x)],[in(3),a(1,y),d(2,y)]),
            [1,a(2,x),c(1,x),[2,a(1,x),[3,c(1,x)]],[4]],R).
       R=[1,a(2,x),c(1,x),[2,[3,a(1,y),c(1,x),d(2,y)]],[4]]

apply(rule(K,_,Mset,[in(NrM)|List]),Ms,R):-
            membrane(K,Ms,Mk),membrane(NrM,Ms,Mnr),
            difference(Mk,Mset,Newk),union(Mnr,List,NewNr),
            transf(K,Newk,Ms,Ms1),transf(NrM,NewNr,Ms1,R),!.
```
◇

rule(K,Mset,[[in(K1),Ob(Freq,y),..],[in(K2),Ob(Freq,y),..],[Ob(Freq,y),..]). This is a little bit more complex. It is a combination of two types of rules. If in membrane K we find the multiset Mset then we take Mset from it and in each membrane K1, K2,.. we put the corresponding list of objects. If the list has no number in front that is there is no in(Kx), then we put the following objects in the same membrane that is K.

Example 8
```
rule(1,3,[a(1,x)],[[in(2),a(1,y)],[b(1,y)]]).
       ?-apply(rule(2,3,[a(1,x)],[[in(3),a(1,y)],[b(1,y)]]),
            [1,d(1,x),[2,a(1,x),[3,c(1,x),d(1,x)]],[4]],R).
       R= [1,d(1,x),[2,b(1,y),[3,a(1,y),c(1,x),d(1,x)]],[4]],R).
       ?-apply(rule(1,4,[a(1,x)],[[in(3),b(1,y)],[in(4),c(1,y)]),
            [1,a(1,x),[2,d(2,x),[3]],[4]],R).
       R= [1,[2,d(2,x),[3,b(1,y)]],[4,c(1,y)]],R).

apply(rule(K,_,Mset,[[H|T]|List]),Ms,R):- membrane(K,Ms,Mk),
            difference(Mk,Mset,D1),transf(K,D1,Ms,Rk),
            collect_union(K,[[H|T]|List],Rk,R),!.
collect_union(K,[],Ms,Ms):-!.
collect_union(K,[[H|T]|List],Ms,RR):- (H=in(I),membrane(I,Ms,Mi),
            union(Mi,T,Newi),transf(I,Newi,Ms,R1),
            collect_union(K,List,R1,RR)),!;
            (not H=in(_),membrane(K,Ms,Mk),union(Mk,[H|T],Newk),
            transf(K,Newk,Ms,R1),collect_union(K,List,R1,RR)).
```
◇

rule(K,RuleNr,Mset,List). This is the case of changing in the same membrane K the multiset Mset with the multiset List.

Example 9
```
rule(4,2,[b(1,x),d(2,x)],[a(1,y),c(2,y)]).
       ?-apply(rule(2,3,[b(1,x)],[c(1,y),d(2,y)]),
            [1,a(2,x),[2,b(1,x),[3,d(1,x)]],[4]],R).
       R=[1,a(2,x),[2,c(1,y),d(2,y),[3,d(1,x)]],[4]],R).
```

```
apply(rule(K,_,Mset,[X|List]),Ms,R):- not X=in(_),not X=out,
        not X=dissolve,membrane(K,Ms,Mk),difference(Mk,Mset,D1),
        union(D1,[X|List],New),transf(K,New,Ms,R),!.
```
◇

As we seen before each apply(rule(MemNr,RuleNr,Mset,List),Ms,R) applies the rule only once. With the predicate try(K,No,Ms,RR) we try to apply the rule(K,No,_,_) on the membrane structure Ms as many times as it is possible. The resulting membrane structure is RR.

try(K,No,Ms,RR). Try rule [K,No] until it is no more applicable on membrane structure Ms.

```
try(K,No,Ms,RR):- rule(K,No,X,Y),apply(rule(K,No,X,Y),Ms,R1),
             write('Rule='),write([K,No]),write(X),write('→'),
             write(Y),nl,write('OLD= '),write(Ms),nl,
             write('SUCCEEDED! New='),write(R1),nl,new(change),
             assert(succeeded(K,No)),try(K,No,R1,RR).
try(K,No,Ms,Ms):- !.
```

The super cell system has a clock. The clock starts with 1 and is incremented by 1. We call a generation the resulting configuration of the membrane structure after we applied all the possible rules in a clock.

```
clock(0).
new(Counter):- Counter(K),X is K+1, retract(Counter(K)),
             assert(Counter(X)).
```

start. This is the main predicate.

```
        ?- start.
        ..  listing from section 4

start:- write('File name for Super Cell= '),read(File),
             consult(File), write('Rules are '),nl,listing(rule),
             write('Order of the rules is'),nl,listing(order),
             write('How many generations?='),read(Gen),
             nl,write('Membrane is '),mstructure(M),write(M),nl,
             again(M,Gen).
again(M,Gen):- new(clock),clock(C),nl,write('Clock='),write(C),
             nl,retractall(change(_)),assert(change(0)),
             retractall(succeeded(_,_)),assert(succeeded(0,0)),
             retractall(tried(_,_)),assert(tried(0,0)),
             write('Membrane='), write(M),nl,
             generation(C,M,R),write('Result='),write(R),nl,
             modify_y(R,Rx),
             (change(X),not X=0, C < Gen, again(Rx,Gen) ; true).
```

We choose only one rule that works successful for a membrane. Therefore, if the rule [1,1] worked we don't try another rule for membrane 1 in this clock. If rule [1,1] does not succeed, we try another rule guided by

```
                order(MembraneNr,RuleNr1,RuleNr2).

   generation(C,M,RR):- rule(K,N,_,_),not succeeded(K,_),
        not tried(K,N),not better_rule(K,N),assert(tried(K,N)),
        try(K,N,M,R),generation(C,R,RR).
   generation(C,M,M):- !.
```

list_of_rules(Rules). In the list Rules we find all the rules of the Super Cell we are working with. We assume in our program that each rule has a number.

```
        ?- list_of_rules(R).
        R=[[1,1],[1,2],[1,3],[1,4],[2,1],[2,2],[3,1],[4,1],[4,2]]

list_of_rules(R):- findall([I,K],rule(I,K,_,_),R).
```

better_rule(K,N). Let's see if rule [K,N] has a better rule in front, that is a rule of higher order that is not tried yet. The answer is yes or no.

```
better_rule(K,N):- order(K,N1,N),not tried(K,N1).
```

modify_x(Membrane_structure, Result). Substitutes all x in the membrane structure by y. We need this in order to simulate the parallelism. We need to unmark the objects (substitute y in x back) when another generation begins.

```
        ?- modify_x([1,a(1,x),[2,b(1,x),c(1,x)],[3]],R).
        R= [1,a(1,y),[2,b(1,y),c(1,y)],[3]]
        ?- modify_y([1,a(1,x),[2,c(1,x),c(1,y)],[3]],R).
        R= [1,a(1,x),[2,b(1,x),c(2,x)],[3]]

modify_y(R,RR):- subst_all(y,x,R,Rx),in_order(Rx,RR).
modify_x(R,RR):- subst_all(x,y,R,Ry),in_order(Ry,RR).
```

4.4 An Example

The program is entirely the collection of predicates presented in section 4. Here is the first example from Păun [1]. The super cell system is described in the file called paun1.dec. This is the listing of the program after we type start.

```
           ?- start.
        File name for Super Cell= paun1.
        rule(1,1,[c(1,x)],[in(4),c(1,y)]).
        rule(1,2,[c(1,x)],[in(4),b(1,y)]).
        rule(1,3,[a(1,x)],[[in(2),a(1,y)],[b(1,y)]]).
        rule(1,4,[d(2,x)],[in(4),a(1,y)]).
        rule(2,1,[a(1,x)],[in(3),a(1,y)]).
        rule(2,2,[a(1,x),c(1,x)],dissolve).
        rule(3,1,[a(1,x)],dissolve).
        rule(4,1,[c(1,x)],[out,d(1,y)]).
        rule(4,2,[b(1,x)],[b(1,y)]).
```

```
          Order of the rules is
       order(1,1,3).
       order(1,2,3).

          How many generations?= 4
Membrane is [1,a(2,x),c(1,x),[2,a(1,x),[3,c(1,x),d(1,x)]],[4]]

   Clock=1
Membrane=[1,a(2,x),c(1,x),[2,a(1,x),[3,c(1,x),d(1,x)]],[4]]
Rule=[1,1][c(1,x)]→[in(4),c(1,y)]
OLD= [1,a(2,x),c(1,x),[2,a(1,x),[3,c(1,x),d(1,x)]],[4]]
SUCCEEDED! New=[1,a(2,x),[2,a(1,x),[3,c(1,x),d(1,x)]],[4,c(1,y)]]
Rule=[2,1][a(1,x)]→[in(3),a(1,y)]
OLD= [1,a(2,x),[2,a(1,x),[3,c(1,x),d(1,x)]],[4,c(1,y)]]
SUCCEEDED! New=[1,a(2,x),[2,[3,c(1,x),d(1,x),a(1,y)]],[4,c(1,y)]]
Result=[1,a(2,x),[2,[3,c(1,x),d(1,x),a(1,y)]],[4,c(1,y)]]

   Clock=2
Membrane=[1,a(2,x),[2,[3,c(1,x),d(1,x),a(1,x)]],[4,c(1,x)]]
Rule=[1,3][a(1,x)]→[[in(2),a(1,y)],[b(1,y)]]
OLD= [1,a(2,x),[2,[3,c(1,x),d(1,x),a(1,x)]],[4,c(1,x)]]
SUCCEEDED!
New=[1,a(1,x),b(1,y),[2,a(1,y),[3,c(1,x),d(1,x),a(1,x)]], [4,c(1,x)]]
Rule=[1,3][a(1,x)]→[[in(2),a(1,y)],[b(1,y)]]
OLD= [1,a(1,x),b(1,y),[2,a(1,y),[3,c(1,x),d(1,x),a(1,x)]],
[4,c(1,x)]]
SUCCEEDED!
New=[1,b(2,y),[2,a(2,y),[3,c(1,x),d(1,x),a(1,x)]],[4,c(1,x)]]
Rule=[3,1][a(1,x)]→dissolve
OLD= [1,b(2,y),[2,a(2,y),[3,c(1,x),d(1,x),a(1,x)]],[4,c(1,x)]]
SUCCEEDED! New=[1,b(2,y),[2,a(2,y),c(1,y),d(1,y)],[4,c(1,x)]]
Rule=[4,1][c(1,x)]→[out,d(1,y)]
OLD= [1,b(2,y),[2,a(2,y),c(1,y),d(1,y)],[4,c(1,x)]]
SUCCEEDED! New=[1,b(2,y),d(1,y),[2,a(2,y),c(1,y),d(1,y)],[4]]
Result=[1,b(2,y),d(1,y),[2,a(2,y),c(1,y),d(1,y)],[4]]

   Clock=3
Membrane=[1,b(2,x),d(1,x),[2,a(2,x),c(1,x),d(1,x)],[4]]
Rule=[2,2][a(1,x),c(1,x)]→dissolve
OLD= [1,b(2,x),d(1,x),[2,a(2,x),c(1,x),d(1,x)],[4]]
SUCCEEDED! New=[1,b(2,x),d(1,x),a(1,y),d(1,y),[4]]
Result=[1,b(2,x),d(1,x),a(1,y),d(1,y),[4]]

   Clock=4
Membrane=[1,b(2,x),a(1,x),d(2,x),[4]]
```

```
Result=[1,b(2,x),a(1,x),d(2,x),[4]]
```

5 Conclusions

This is the first version of the program. Let's call it ProMem 0.1, from *Prolog for Membranes*. The program is entirely presented in the section 4 and is written in LPA [6]. We will highly appreciate any comments concerning the program because as any first version, it might have bugs.

In writing ProMem 0.1, we had in mind only the transparency of the code in order to follow the features of membrane computing paradigm. We did not try to make programming shortcuts or tricks that might have given an optimal program. Our intention was to write a program so transparent that anyone who knows Prolog can understand how a super cell system works and any person familiar with the super cell system could read the Prolog program.

ProMem 0.1 is devoted to develop applications for this new computational paradigm in order to evaluate the power and the opportunity to build actual machines.

It might also be useful for the designers of this new paradigm of computation in order to "play" with all kinds of rules for the super cell systems. In this sense we think that the next version should have graphics to visualize the mobility of the objects.

References

[1] Gheorghe Păun: *Computing with Membranes*, Turku Centre for Computer Science, TUCS Technical report, N.208, November 1998 (www.tucs.fi) and *Journal of Computer and System Sciences*, 61 (2000).

[2] Jurgen Dassow, Gh. Păun: "On the Power of Membrane Computing", *FCT'99*, Iaşi, Romania, 1999.

[3] Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa: "Membrane Computing with External Output", *FCT'99*, Iaşi, Romania, 1999.

[4] Ivan Bratko: *PROLOG, Programming for Artificial Intelligence*, Addison-Wesley Pub. Comp, 1990.

[5] Dave Westwood: *LPA-Prolog 2.6 Technical reference*, LPA Ltd, London England,1994.

[6] Gh. Păun, Computing with membranes. An introduction, *Bulletin of the EATCS*, 67 (Febr. 1999), 139–152.

[7] Gh. Păun, Computing with membranes – A variant: P systems with polarized membranes, *Intern. J. of Foundations of Computer Science*, 11, 1 (2000), 167–182, and *Auckland University, CDMTCS Report* No 098, 1999 (www.cs.auckland.ac.nz/CDMTCS).

[8] Gh. Păun, P systems with active membranes: Attacking NP-complete problems, *J. Automata, Languages and Combinatorics*, to apear, and *Auckland University, CDMTCS Report* No 102, 1999 (www.cs.auckland.ac.nz/CDMTCS).

[9] Gh. Păun, Computing with membranes. A correction, two problems, and some bibliographical remarks, *Bulletin of the EATCS*, 68 (1999), 141–144.

[10] Gh. Păun, Computing with membranes (P Systems); Attacking NP-complete problems, *Unconventional Models of Computing* (I. Antoniou, C. S. Calude, M. J. Dinneen, eds.), Springer-Verlag, 2000 (in press).

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 176 - 190.

Monoidal Systems and Membrane Systems

Vincenzo Manca
Università di Pisa
Dipartimento di Informatica
Corso Italia, 40 - 56125 Pisa - Italy
e-mail: mancav@di.unipi.it

Keywords and Phrases: String Rewriting, Formal Systems, Formal Languages, Membrane Systems, Logical Representability.

Abstract

Monoidal systems are introduced that are computationally universal formalisms where a great quantity of other formalisms can be easily represented. Many particular symbolic systems from different areas are expressed as monoidal systems. The possibility is outlined that these systems express localization aspects typical of membrane systems and other phenomena such as temporality and multiplicity that are essential in the formalization of molecule manipulation systems.

1 Introduction

In [12] we introduced some forms of logical representability in the study of string derivation systems, giving examples of logical representations for many kinds of symbolic systems, and general theorems showing the computational universality of different forms of logical representability, their relationships, and their applicability in the analysis and unification of many classical formalisms.

Among the methods of logical representations for symbolic systems, monoidal theories and monoidal representability result to be very expressive tools. In [13] some normalization results were presented in the general context of string derivation and some monoidal theories were given in connection with some regulation mechanisms and with some examples of complex systems taken from L-systems and grammar systems areas.

In this paper we continue to study monoidal representability by introducing *monoidal systems*. In short, a monoidal system M consists of an alphabet A, a finite set P of predicates, some finite axioms Φ plus the monoid axioms and a subset Q of P, called *representation* predicates. The signature of the axioms consists of: i) the symbols for the monoid operation and the monoid unit (concatenation and λ), ii) the predicates P, and iii) the symbols of A as individual constants. A k-ary representation predicate S defines a k-ary relation R over A^* such that (strings are closed terms): $R(\alpha_1, \ldots \alpha_k) \iff \Phi \models S(\alpha_1, \ldots \alpha_k)$. This simple idea allows us to apply the first order logical apparatus for expressing: rewriting relations, derivations, regulations, and other concepts typical of symbolic systems. It is enough to axiomatize by suitable predicates the structure of the system we want describe: all the dynamical aspects of the systems can be *deduced*, with a logical calculus, from these axioms. In many cases it is natural to distinguish the axioms common to all the systems of a class (grammars, automata, transducers, ...) from the axioms that are proper of a particular system. In the sequel, we will present many examples aimed at showing the large spectrum of applicability of monoidal systems. In a special manner, we want to stress the intrinsic potentiality of monoidal systems in expressing phenomena where some forms of localization mechanisms are considered: membranes, environments, or regions. In a series of papers concerning the logical formalization of biochemical phenomena we already formulated in several forms some ideas of possible formalizations of localization principles (see the *metabolic model META* in [9], and the rules of *logical metabolic systems* in [10, 12]), however, in the course of these attempts we arrived to the general conclusion that any formal system able to cope with molecule manipulation systems, arising in biochemical contexts, have to develop tools for describing not only string generation or recognition, but also more general dynamical aspects: *locality* (interactivity, osmosis), *temporality* (stability, periodicity), and *multiplicity* (growth, energetic trade-off). In other words, space, time, and matter/energy aspects have to be accounted for in any satisfactory modelling of dynamical systems based on molecules.

P systems introduced in [16] and developed in many other papers (for example [17, 18]) are systems explicitly devoted to a formalization of localization phenomena in string elaborations. The membrane structure of a P system can be easily expressed by suitable predicates and axioms; moreover, many different structural choices, and regulation strategies within these systems can be formulated by other axioms. A monoidal system related to P systems will be given in a next section.

We think that monoidal systems could be a good basis for developing systems where not only locality, but also temporality, and multiplicity can be dealt with in a very general way. In fact, locality is given for free just by the use of predicates, and temporality can be easily introduced by predicates with a temporal parameter. Multiplicity can be represented by strings or by a typing predicate ($x : \alpha$ means that x is an individual whose type is represented by the string α).

Of course, this is only a starting point. In fact, many physical aspects (polarity, osmosis, energetic trade-off, ...) could require more specific representation tools; however, it seems us that monoidal systems have an intrinsic flexibility that allows us to extend in many directions their potentialities. Our motto for future research is: *molecules are strings with additional features intrinsic to their physicality; find predicates and axioms suitable to express this physicality in the right manner (for a wide spectrum of situations).*

In the sequel, we refer to [22, 21] for basic elements of formal language theory, and to [3, 23] for basic elements of mathematical logic.

2 Monoidal Systems

We recall that:

$$\Phi \models \varphi$$

means that Φ is a Σ-theory over a signature Σ, φ is a formula on the same signature, and φ is a logical consequence of the theory Φ (φ holds in all the first order Σ-models $\mathcal{M}$ of Φ).

Given a formula $\varphi(x)$ with a free variable x and an individual term t, we indicate by $\varphi(t)$ the formula obtained from $\varphi(x)$ by replacing in it all the occurrences of x with the individual term t (if in t some variable occurs, then it has to be free in $\varphi(t)$).

Let A be a finite alphabet. A *monoidal signature*, of alphabet A and predicates P, consists of: i) the symbols of A as constants plus another constant λ for the empty string, ii) a binary function symbol for concatenation (which we indicate by juxtaposition), and iii) a finite set P of symbols for predicates.

Definition 2.1 *A* monoidal *theory of alphabet A and predicates P is a theory over a monoidal signature (of alphabet A and predicates P) that includes the usual axioms of* monoid *(the associativity of concatenation and the indifference of λ with respect to concatenation).*

Of course, in a monoidal theory the set of closed terms (terms without variables) consists of the free monoid A^*.

We call *proper axioms* of a monoidal theory the axioms that are different from the monoid axioms. Smullyan's formal systems [24] are particular case of monoidal theories.

Definition 2.2 *Let Φ be a monoidal theory over a signature Σ, and let $\varphi(x)$ be a Σ-formula with only one free variable. A language over the alphabet A is representable in Φ by the formula φ if:*

$$\alpha \in L \iff \Phi \models \varphi(\alpha).$$

Example 2.1 $\{a^n b^n c^n | n \in \omega\} \subseteq \{a, b, c\}^*$ *is representable by L in the the monoidal theory having the following proper axioms:*

1. $L(\lambda)$
2. $L(abc)$
3. $\forall x\ y\ (L(xby) \rightarrow L(axbbyc))$.

For example, this is the way we deduce $L(aabbcc)$.

1. $L(abc)$ *axiom 2*
2. $\forall x\ y\ (L(xby) \rightarrow L(axbbyc))$ *axiom 3*
3. $L(abc) \rightarrow L(aabbcc)$ *instance of axiom 3 for $x = a, y = b$*
4. $L(aabbcc)$ *modus ponens from 1, 3.*

Assume that $\Phi \models L(a^n b^n c^n)$, then by the third axiom for $x = a^n b^{n-1}, y = c^n$ we get:
$L(a^n b^n c^n) \rightarrow L(aa^n b^n bc^n c)$, *and by modus ponens:* $\Phi \models L(a^{n+1} b^{n+1} c^{n+1})$, *therefore:*

$$\alpha \in \{a^n b^n c^n | n \in \omega\} \iff \Phi \models L(\alpha).$$

Definition 2.3 *A monoidal system M of alphabet A, axioms Φ, predicates P, and representation predicates Q is a system*

$$M = (A, P, \Phi, Q)$$

where Φ are the proper axioms of a monoidal theory of alphabet A and predicates P, and Q is a subset of P.

A *monoidal grammatical system* M is a monoidal system with a representation unary predicate; if L is its representation predicate, then M defines a language $L(M)$ given by the language representable in the monoidal theory of M by the predicate L.

Let C be the grammatical monoidal system of alphabet $\{a, b, c\}$, with the representation predicate L and with the axioms given in the previous example, then $\{a^n b^n c^n | n \in \omega\} = L(C)$.

In the following we use lower case letters for the symbols of the alphabet A, capital letters or strings beginning with capital letters for predicates, and strings ending with $*$ for representation predicates. In this manner a monoidal system can be completely expressed by its axioms (variables will be specified explicitly).

The class ML consists of the languages defined by means of monoidal grammatical system. The following theorem establishes the computational universality of monoidal systems.

Theorem 2.1 *(Universality of Monoidal Systems) $RE = ML$.*

Proof. We show that for any Chomsky grammar $G = (A, T, S, R)$, where A is the alphabet of G, T the terminal symbols of G, S the start symbol of G, and R the productions of G, we can define a monoidal grammatical system M_G of alphabet A and predicates $\{Start, Derive, Replace, Terminal, Generate*\}$ such that $L(G) = Generate*(M_G)$. The system M_G is given by the following axioms (x, y, u, v variables implicitly universally quantified; S, a, α, β closed terms):

- $Start(x) \to Derive(x)$
- $Derive(uxv) \wedge Replace(x, y) \to Derive(uyv)$
- $Derive(x) \wedge Terminal(x) \to Generate*(x)$
- $Terminal(x) \wedge Terminal(y) \to Terminal(xy)$
- $Start(S)$
- $Terminal(a) \quad \forall \;\; a \in T$
- $Replace(\alpha, \beta) \quad \forall \;\; \alpha \to \beta \in R.$

It follows easily by induction that a terminal string α is generated by G iff the formula $Generate*(\alpha)$ is deduced by the given axioms. This implies that $RE \subseteq ML$, the converse inclusion is a consequence of a general theorem about axiomatic systems: the theorems of an axiomatic theory are a recursively enumerable set [23]. *Q.E.D.*

The example given for the language $\{a^n b^n c^n | n \in \omega\}$ shows that a monoidal system for a given language can be more easily defined in a direct manner rather than by the system M_G associated to a grammar G that generates the language.

The proof of the previous theorem gives an important information about the logic we need in defining monoidal grammatical systems: it is not all first order logic, but only a part of it, usually indicated as *Horn logic*. For this logic we have a simple logical calculus $\vdash$ in order to deduce all the logical consequences of some axioms Φ. Namely, the axioms of a monoidal grammatical system are universal quantifications of atomic formulae, of conjunctions of atomic formulae, or of implications between a conjunction of atomic formulae and an atomic formula. In this case we have that:

$$\Phi \models \varphi \iff \Phi \vdash \varphi$$

and $\vdash$ can be defined by these simple deductive rules (t any term):

- $\varphi \in \Phi \Rightarrow \Phi \vdash \varphi$
- $\Phi \vdash \varphi, \Phi \vdash \psi \Rightarrow \Phi \vdash \varphi \wedge \psi$
- $\Phi \vdash \varphi \rightarrow \psi, \Phi \vdash \varphi \Rightarrow \Phi \vdash \psi$
- $\Phi \vdash \forall x \varphi(x) \Rightarrow \Phi \vdash \varphi(t)$.

In [12] we proved that a language is representable in a monoidal theory iff it is representable, in the model SEQ of finite sequences of natural numbers with concatenation and length, by means of Σ_1-SEQ formulae (a particular class of $\forall$-bounded formulae). Another interesting aspect, resulting from the proof of the universality theorem above, is that the axioms of the monoidal system related to a grammar G can be divided in two parts: a *general* part (the first 4 axioms) are common to any monoidal system associated to a Chomsky grammar, while the other *particular* axioms (the last 3 axioms) are relative to the grammar G. It is very simple to find monoidal systems for many classes of formalisms studied in formal language theory (e.g., L-systems, H-systems, [8, 6, 5]); examples essentially based on monoidal systems can be found in [12, 13]. Now we consider finite state automata and finite iterated transducers [25, 20, 11], where the division into general and particular axioms is completely apparent.

Example 2.2 *(Monoidal Systems for Finite Automata)*
Let (A, Q, q_0, F, R) be a finite state automaton of alphabet A, states Q, initial state q_0, final states F, and transition rules R. The following are the axioms of a grammatical monoidal system M such that Recognize$*(M)$ is the language recognized by the automaton (x, y, z, t, u, v, w variables implicitly universally quantified; q, q_0, s, a, α closed terms):

- $Input(x) \wedge Input(y) \rightarrow Input(xy)$
- $Input(x) \wedge Initial(z) \rightarrow Derive(zx)$
- $Derive(uzxv) \wedge Transition(zx, t) \rightarrow Derive(uxtv)$
- $Derive(wz) \wedge Final(z) \rightarrow Recognize{*}(w)$
- $Input(a) \quad \forall \;\; a \in A$
- $Initial(q_0)$
- $State(q) \quad \forall \;\; q \in Q$
- $Final(q) \quad \forall \;\; q \in F$
- $Transition(qa, s) \quad \forall \;\; qa \rightarrow s \in R.$

Example 2.3 *(Monoidal Systems for Iterated Transducers)*
Let (A, Q, q_0, a_0, F, R) be a finite iterated transducer of alphabet A, states Q, initial state q_0, initial symbol a_0, final states F, and transition rules R. The following are the axioms of a grammatical monoidal system M such that Generate$*$ (M) is the language generated by the transducer (x, y, z, t, u, v, w variables implicitly universally quantified; $q, q_0, s, a_0, a, \alpha$ closed terms):

- $Start(x) \wedge Initial(z) \rightarrow Derive(zx)$
- $Derive(uzxv) \wedge Transition(zx, yt) \rightarrow Derive(uytv)$
- $Derive(wz) \wedge State(z) \wedge Initial(t) \rightarrow Derive(tw)$
- $Derive(wz) \wedge Final(z) \rightarrow Generate{*}(w)$

- $Start(a_0)$
- $Initial(q_0)$
- $State(q) \quad \forall \ q \in Q$
- $Final(q) \quad \forall \ q \in F$
- $Transition(qa, \alpha s) \quad \forall \ qa \to \alpha s \in R.$

3 A Monoidal System for Red Algae

Red Algae are a famous example of application of L-systems in the formalization of developmental processes. In the usual *turtle* representation, the first six growth stages of a red alga are the following (F is a cell, drawn as a segment, what is between brackets is alternatively at a positive or negative angle with respect to the main growth axis):

- $R(1) = F$
- $R(2) = FF$
- $R(3) = FFFF$
- $R(4) = FF[F]FF$
- $R(5) = FF[FF]FF[F]FF$
- $R(6) = FF[FFF]FF[FF]FF[F]FFFF$

The growth process continues according to the following procedure (see [6]):

> "From stage 6 onwards we may divide the organism into two parts. The first six cells (from the left) of the main branch form a basal part while the rest of the cells forms an apical part. Every second cell in the basal part carries a non-branching filament. These filaments develop linearly in time, they repeat at each stage their own previous structure with the addition of one new cell. At stage 6 the lengths of these filaments are 3, 2, 1, respectively, the longer ones being nearer the base. The apical part at stage 6 consists of four cells without any branches. After this, the apical part at each stage is a repeat of the apical part of the previous stage, together with two new cells at the base end of the apical part. The second of these new cells carries a branch which is identical to the whole organism six stages previously."

Formal representations of this development in terms of OL systems can be found in [8, 22, 6, 2, 7].

Here we present a monoidal theory that is a natural translation of the informal description given above. In fact, for $n > 6$ we have the following conditions, where $R(n), B(n), A(n)$ are the strings representing the entire organism, the basal part, and the apical part respectively at stage n (x, y, z are variables implicitly universally quantified, n can be represented by the string of n symbols F):

- $A(6) = FFFF$
- $R(n) = B(n)A(n)$
- $A(n) = FF[R(n-6)]A(n-1)$
- $y \subset xyz$

- $[x] \subset B(n-1) \rightarrow [xF] \subset B(n)$.

(By the way, it is easy to provide a generalized sequential mapping g such that for any $n \geq 6$, $g(R(n)) = a^n b^n c^n$; in fact, the basal part has a threefold synchronized development, therefore Red Algae are not a context free language). From a technical viewpoint, the axioms given above are not a monoidal theory because they include terms different from the strings of a free monoid. However, this is only matter of syntactic sugar: it is very easy to transform these axioms in the right form by transforming the functional symbols R, A, B into predicates. We prefer this presentation because it makes more evident the translation from the informal definition of the growth process.

4 A Monoidal System for Proteins

In [19] Pawlak introduced a formal language as an attempt to formalize the process of protein formation. Since there are 64 types of codons (strings of length 3 over the nucleotide alphabet $\{0, 1, 2, 3\}$), but only 20 of them are associated with some amino acid, Pawlak selects those codons that can be associated to some particular triangles representing amino acids, that are just 20, and gives a recursive definition of proteins, as the *well-formed* strings resulting from this definition. What it is interesting, from our point of view, it is not the biochemical adequacy of this language, but the fact that it is a language not easily definable with the usual tools of formal language theory. Definitions of this language in terms of Chomsky grammars were proposed, but the equivalence of these definitions with the original definition of Pawlak is not completely obvious (a discussion on this regard, in the more general context of formal languages as models of genetics, can be found in [14, 15]). The initial idea of Pawlak is a linear ordering relations over the four bases A, T, C, G (this is the reason we indicate them by $0, 1, 2, 3$). The restriction proposed by Pawlak is that an amino acid is represented by by a triangle labelled by the symbols of a codon ijk such that if i is the label of the left left side, j the label of the base, and k the label of the right side, then $i < j \geq k$. It easy to see that there are 20 triangle satisfying this condition, that we say *amino triangles*. The recursive definition of Pawlak's language is the following:

- Every amino triangle is a well-formed polytriangle;
- Given a polytriangle x, we get a new polytriangle if we add to x an amino triangle such that: its base and the relative label coincide either with the left side and the relative label of a triangle of x, or with the right side and the relative label of a triangle of x, and no side of the added triangle may coincide with the base or the side of another triangle of x;
- A polytriangle is *terminal* if no amino triangle can be added to it that gives a new polytriangle;
- A *protein* is a terminal polytriangle.

Pictorial representations of this language can be found in [14]. In the following we give a monoidal system which in a very natural manner defines strings which represents proteins according to this definition.

- $0 < 1 < 2 < 3$
- $x = 0 \vee x = 1 \vee x = 2 \vee x = 3 \rightarrow Base(x)$
- $Base(x) \wedge Base(y) \wedge Base(z) \wedge x < y \wedge (z = y \vee z < y) \rightarrow Amino(xyz)$

- $Amino(x) \rightarrow Polypeptide(x)$
- $Polypeptide(wxyz) \wedge Amino(xyz) \wedge Amino(uzv) \rightarrow Polypeptide(wxyzuzv)$
- $Polypeptide(xyzw) \wedge Amino(xyz) \wedge Amino(uxv) \rightarrow Polypeptide(uxvxyzw)$
- $Polypeptide(xyzwutv) \wedge Amino(xyz) \wedge Amino(uxv) \wedge x = 0 \wedge v = 0 \rightarrow Terminal(xyzwutv)$
- $Polypeptide(w) \wedge Terminal(w) \rightarrow Protein*(w)$

(a triangle is a string of 3 symbols that are the labels of left side, base, and right side; when two of these strings are contiguous, then the same symbols occurring in them will represent labels that share the same segment).

5 A Monoidal System for Primality

In [10] a membrane system for the generation of prime numbers was given that realizes a multi-agent version of Eratosthenes' sieve. It is essentially based on the following recursive definition of the function $q(n)$ giving the value of the greatest prime number not exceeding n:

$$q(2) = 2$$

and for $n > 2$:

$$q(n) = min\{i \leq n \mid i \neq j \cdot q(k) \quad \forall \;\; 1 < j, k < n\}$$

(for $n > 1$, $q(n) = n$ if and only if n is a prime number). In the aforementioned system the sieve was structured by generating natural numbers and by adding a new membrane when a prime number was discovered, in such a way that in any membrane, labeled with a prime, at any time there is included the biggest multiple of its label among those generated at that time.

This process can be expressed by the following monoidal system where a temporal parameter is indicated as a prefix of predicates ($Mult(z, y)$ means that z is a multiplier of y; the meaning of the other predicates is obvious; x, y, z, u, v are implicitly universally quantified variables).

- $0\text{–}Prime(2)$
- $0\text{–}Current(3)$
- $0\text{–}Mult(4, 2)$
- $j\text{–}Current(x) \wedge (j\text{–}Mult(z, y) \rightarrow x < z)$
  $\rightarrow (j + 1)\text{–}Prime(x) \wedge (j + 1)\text{–}Current(x + 1) \wedge (j + 1)\text{–}Mult(x + x, x) \wedge (j + 1)\text{–}Mult(z, y)$
- $j\text{–}Current(x) \wedge j\text{–}Mult(x, y) \wedge j\text{–}Mult(u, v) \wedge u \neq x$
  $\rightarrow (j + 1)\text{–}Current(x + 1) \wedge (j + 1)\text{–}Mult(u, v) \wedge (j + 1)\text{–}Mult(x + y, y)$
- $j\text{–}Prime(x) \rightarrow Prime*(x)$

The following is what is deduced for the first seven steps:

$$0\text{–}Prime(2), 0\text{–}Current(3), 0\text{–}Mult(4, 2)$$

$$1\text{–}Prime(3), 1\text{–}Current(4), 1\text{–}Mult(4, 2), 1\text{–}Mult(6, 3)$$

$$2\text{–}Current(5), 2\text{–}Mult(6, 2), 2\text{–}Mult(6, 3)$$

$$3\text{–}Prime(5), 3\text{–}Current(6), 3\text{–}Mult(6,2), 3\text{–}Mult(6,3), 3\text{–}Mult(10,5)$$

$$4\text{–}Current(7), 4\text{–}Mult(8,2), 4\text{–}Mult(9,3), 4\text{–}Mult(10,5)$$

$$5\text{–}Prime(7), 5\text{–}Current(8), 5\text{–}Mult(8,2), 5\text{–}Mult(9,3), 5\text{–}Mult(10,5), 5\text{–}Mult(14,7)$$

$$6\text{–}Current(9), 6\text{–}Mult(10,2), 6\text{–}Mult(9,3), 6\text{–}Mult(10,5), 6\text{–}Mult(14,7).$$

We remark that the only arithmetic operation we need in this generative process is the sum; this means that any implementation of the sum via DNA is a good basis for a DNA process of prime generation [4].

6 Monoidal Systems for SAT

In [18] a solution of the problem SAT is given by using a P system, in a time which is linear in the number of variables and of clauses. The idea of this method is the following. Let $v_1, \dots v_n$ be n propositional variables and let $C_1, \dots C_m$ be m clauses over the given propositional variables. Create m nested membranes labelled with the clauses inside an external membrane labelled by C_0 (C_0 *includes* C_1 that *includes* C_2, ..., that *includes* C_m). Inside the most internal membrane, labelled by C_m, put the string $v_1, \dots v_n$ of all variables. Then, duplicate this membrane into two membranes, both labelled by C_m, where the strings $t_1, \dots v_n$, and $f_1, \dots v_n$ are put respectively. Continue this duplication process for all the membranes labelled by C_m, for all the propositional variables, until every variable v_i is replaced by the two corresponding truth values t_i, and f_i. At any step a membrane labelled by C_m produces two membranes labelled by C_m, therefore after n steps all the possible 2^n Boolean valuations of the n propositional variables are generated into m membranes labelled by C_m. All these membranes are inside a membrane labelled by C_{m-1}, that is inside a membrane C_{m-2}, an so on, being C_0 the most external membrane. At this point a process of membrane dissolving is performed by using the clauses. Starting from the most internal membranes (labelled by C_m), if a valuation contains t_i and the clause which labels the membrane contains the literal v_i, then the membrane is dissolved; while, if a valuation contains f_i and the clause contains the literal $\neg v_i$ then the membrane is dissolved. In this manner, some valuation reaches the external membrane if the given set of clauses is satisfiable. This process can be represented by the following monoidal theory (x, y variables). The essence of linearity of this solution is intrinsically related to the second axiom. Any computational model where this deduction can be represented in a parallel way can also generate all the valuations in $O(n)$ time. Moreover, if we could activate, in a massive parallel way, a test procedure for the dissolution of membranes including each generated valuation, we can completely solve the problem in $O(n+m)$ time.

- $Variables(v_1 v_2 \dots v_n)$
- $Variables(xv_iy) \rightarrow Variables(xt_iy) \wedge Variables(xf_iy)$
- $Valuation(t_i) \wedge Valuation(f_i)$
- $Valuation(x) \wedge Valuation(y) \rightarrow Valuation(xy)$
- $Valuation(x) \wedge Variables(x) \rightarrow Test_m(x)$
- $Clause_j(xv_iy) \wedge Test_j(xt_iy) \rightarrow Test_{j-1}(xt_iy)$
- $Clause_j(x\neg v_iy) \wedge Test_j(xf_iy) \rightarrow Test_{j-1}(xf_iy)$

- $Test_0(x) \rightarrow Solution{*}(x)$.

Now we give a monoidal theory where a solution of SAT is constructed without a preliminary generation of all possible valuations of propositional variables $(0, 1$ stand for the truth values and ? for any of them; x, y, u, v are variables). Encode the m clauses with the strings $\beta_1\beta_2 \ldots \beta_m$. Encode $v_1, \ldots, v_n$ with with the strings $\alpha_1, \alpha_2 \ldots, \alpha_n$. The first axioms are put for any value i $(1 \leq i \leq n)$ such that $\beta_{i_1}\beta_{i_2} \ldots \beta_{i_k}$ encode the k clauses satisfied by $v_i = true$. The second axioms are put for any value i $(1 \leq i \leq n)$ such that $\beta_{j_1}\beta_{j_2} \ldots \beta_{j_h}$ encode the h clauses satisfied by $v_i = false$. The third axioms are put for the values of i such that no clause is satisfied by $v_i = true$ or by $v_i = false$.

1. $Positive_i(\alpha_i 1 \beta_{i_1}\beta_{i_2} \ldots \beta_{i_k})$
2. $Negative_i(\alpha_i 0 \beta_{j_1}\beta_{j_2} \ldots \beta_{j_h})$
3. $Neutral_i(\alpha_i ?)$
4. $Positive_i(x) \rightarrow Component_i(x)$
5. $Negative_i(x) \rightarrow Component_i(x)$
6. $Neutral_i(x) \rightarrow Component_i(x)$
7. $Component_i(x) \rightarrow Component_i(uxv)$
8. $Component_1(x) \wedge \ldots \wedge Component_n(x) \rightarrow Presolution(x)$
9. $Substring(x, uxv)$
10. $Presolution(x) \wedge Substring(\beta_1, x) \wedge \ldots \wedge Substring(\beta_m, x)$
    $\rightarrow Solution{*}(x)$.

The following (abstract) DNA solution of SAT was suggested by the previous monoidal system. Encode the m clauses with the oligonucleotides $\beta_1\beta_2 \ldots \beta_m$ (strings in the alphabet $\{T, A, C, G\}$). Encode $v_1, \ldots, v_n$ with with the oligos $\alpha_1, \alpha_2 \ldots, \alpha_n$. Construct for $i = 1, 2, \ldots n$, the oligo

$$Positive_i = \alpha_i T \beta_{i_1}\beta_{i_2} \ldots \beta_{i_k}\overline{\alpha_{i+1}}$$

if $\beta_{i_1}\beta_{i_2} \ldots \beta_{i_k}$ encode the k clauses satisfied by $v_i = true$, where $\overline{\alpha_{i+1}}$ is the oligo complementary of α_i. Construct analogously the oligo

$$Negative_i = \alpha_i A \beta_{j_1}\beta_{j_2} \ldots \beta_{j_h}\overline{\alpha_{i+1}}$$

if $\beta_{j_1}\beta_{j_2} \ldots \beta_{j_h}$ encode the h clauses satisfied by $v_i = false$. Moreover, put the oligo

$$Neutral_i = \alpha_i C \overline{\alpha_{i+1}}$$

if no clause is satisfied by $v_i = true$ or by by $v_i = false$.

Amplify with PRC and add ligase for allowing complementary strands to anneal. Filter by affinity, in m consecutive steps, the DNA strands where all $\beta_1\beta_2 \ldots \beta_m$ occur. If some strand remains in the final test tube, then it encodes a solution of the given instance of SAT: it is enough sequencing the resulting DNA strand, the base that follows α_i indicates the right value to associate to the variable v_i (T stands for $true$, A for false, and C for any truth value).

7 Monoidal Systems for P Systems

P systems [16, 17, 18] are membrane systems where aspects of locality, and multiplicity are explicitly treated. We show that a monoidal representation of these systems is strongly related to an explicit introduction of some temporal parameters.

The logical description we propose automatically fulfils some general requirements that are requested to their dynamics. Here we follow the variant presented in [18]. Our starting point is the term $[_h\ a\ ]_h^p$ that expresses a membrane of label h and polarity p where the object a is located. In a P system an inclusion structure of labelled membranes is given and multisets of objects (represented as strings) are located inside them. The behavior of such a system is driven by six types of rules:

- evolution rules (an object inside a membrane is changed);
- introduction rules (an object outside a membrane is put inside it);
- extraction rules (an object inside a membrane is sent outside it);
- dissolution rules (a membrane disappears and the objects inside it are sent outside);
- elementary division rules (a membrane transforms into two membranes possibly changing the polarity and replacing an object with other two in the new membranes);
- division rules (submembranes with opposite polarizations are separated into two new membranes, possibly changing their polarities).

Let us consider:

- three ternary predicates: $\in, :, \subset$ and two binary predicates $<, \downarrow$;
- individual constants for labels, membranes, polarities, and object occurrences (M_0 is the label for the most external membrane; E a constant for the external environment; $+, -, 0$ for polarities, and 0 also for the initial instant);
- the concatenation operator for expressing multisets by strings (all the permutations of a string α represent the same multiset where an object has n occurrences if the correspondent symbol occurs n times in α);
- a next step operator (t' is the step that follows t).

The meaning of these predicates is the following:

- $(p)m :_t h$
  "at step t the membrane m has the label h and the polarity p";
- $xay \in_t m$
  "at step t an occurrence of the object a is located in the membrane m";
- $m_1 \subset_t m_2$
  "at step t the membrane m_1 is included in the membrane m_2";
- $t_1 < t_2$
  "the instant t_1 is before the instant t_2";
- $m \downarrow t$
  "in the instant t the membrane m changed with respect to the previous instant" ($t > 0$, the initial configuration will refers to the initial instant 0).

Now the rules of a P system can be easily expressed in the following manner where $m \uparrow t$ stands for $\forall x(m \downarrow x \rightarrow x < t)$ and a notation such as

$$(t := t')[F(m,t) - \ xay \in_t m]$$

stands for the set of formulae already deduced where m, t occur, minus the formula $xay \in_t m$, after replacing t with t' (x, y, z, t are implicitly universally quantified variables, and the comma will abbreviates the logical conjunction).

- $m \uparrow t\ ,\ xay \in_t m \rightarrow xwy \in_{t'} m\ ,\ (t := t')[F(m,t) - \ xay \in_t m]$ (evolution rule)
- $m_1 \uparrow t\ , m_2 \uparrow t\ , xay \in_t m_1\ , z \in_t m_2\ , m_2 \subset_t m_1 \rightarrow za \in_{t'} m_2\ , xy \in_{t'} m_1,$ $(t := t')[F(m_1, m_2, t) - \ xay \in_t m_1, - \ z \in_t m_2]$ (introduction rule)
- $m_1 \uparrow t\ ,\ m_2 \uparrow t\ ,\ xay \in_t m_2\ ,\ z \in_t m_1\ ,\ m_2 \subset_t m_1 \rightarrow xy \in_{t'} m_2\ ,\ za \in_{t'} m_1, (t := t')[F(m_1, m_2, t) - \ xay \in_t m_2, - \ z \in_t m_1]$ (extraction rule)
- $m_1 \uparrow t\ ,\ m_2 \uparrow t\ ,\ xay \in_t m_2\ ,\ m_2 \subset_t m_1 \rightarrow\ ,\ xby \in_{t'} m_1,$
  $(t := t', m_2 := m_1)[F(m_1, m_2, t) - \ xay \in_t m_2 - \ m_2 \subset_t m_1]$
  (dissolution rule)
- $m_1 \uparrow t\ ,\ m_2 \uparrow t\ ,\ (p)m_2 :_t h\ ,\ xay \in_t m_2\ ,\ m_2 \subset_t m_1 \rightarrow$
  $m_3 \subset_{t'} m_1\ ,\ m_4 \subset_{t'} m_1\ ,\ xby \in_{t'} m_3\ ,\ xcy \in_{t'} m_4,$
  $(q)m_3 :_{t'} h\ ,\ (s)m_4 :_{t'} h,$
  $(t := t', m_2 := m_3)[F(m_1, m_2, t) - \ m_2 \subset_t m_1 - \ (q)m_2 :_t h \ - xay \in_t m_2]$
  $(t := t', m_2 := m_4)[F(m_1, m_2, t) - \ m_2 \subset_t m_1 - \ (q)m_2 :_t h \ - xay \in_t m_2]$
  (elementary division rule)
- $m \uparrow t\ ,\ m_1 \subset_t m\ ,\ \ldots, m_k \subset_t m\ ,\ \ldots, m_j \subset_t m, \ldots, m_n \subset_t m, (p)m :_t h,$
  $(+)m_1 :_t h_1\ ,\ \ldots, (+)m_k :_t h_k,$
  $(-)m_{k+1} :_t h_{k+1}\ ,\ \ldots, (-)m_j :_t h_j,$
  $(0)m_{j+1} :_t h_{j+1}\ ,\ \ldots, (0)m_n :_t h_n \ \rightarrow$
  $(t := t', m := m_a, p := p_a)[F(m,t) - m_{k+1} \subset_t m, \ldots, m_j \subset_t m],$
  $(t := t', m := m_b, p := p_b)[F(m,t) - m_1 \subset_t m, \ldots, m_k \subset_t m]$
  (division rule, $m \neq M_0$).

In this monoidal representation of P systems, a crucial point is the use of temporal parameters. In fact, there is a deep difference between the intrinsic monotony of classical logic and the nonmonotony of a system where during the process some element can disappear. In our rules we put in the premises of implications the *last instant requirement* ($m \uparrow t$) and in the conclusion the *temporal updating* where some facts are not updated, and so cannot anymore determine the future behavior of the system. We remark that in this formulation time is local to membranes. This makes possible having deductive processes that are mutually independent. But, in order to partaking to the application of rules, membranes that are in the inclusion relationship need to have the same *last instant*. This require an additional *improper* rule of *pure evolution*:

$$\varphi(t) \rightarrow \varphi(t')$$

for every already deduced formula $\varphi(t)$ (this rule could be replaced by more complex conditions in the rules given above). A membrane that does not apply this rule in an appropiate way may block the application of rules giving parasitic behaviors.

Many variants can be considered in dealing with the temporal parameter, where more sophisticated temporal relations can be combined with localization principles. Here, we do not go on in this analysis, but it is important to realize that the monoidal formalism provides us a tool adequate for these investigations.

A P system is a generative device where a string α is an output of the system if at end of a computation, when no rules can be applied, $\alpha \in E$ (α represents the multiset in the external environment where any order can be chosen when many objects are sent out at the same time). The language generated consists of all outputs corresponding to all initial configurations of the system.

In the usual manner we did for the other formalisms, if PP are the axioms given above, and $Th(M)$ are the axioms specifying the membrane structure of a particular P system M with an initial configuration, we have:

$$PP \cup Th(M) \models \alpha \in_t E$$

if and only if α is an output of M in correspondence to the given initial configuration, and no proper rule can be applied anymore (t is a final instant).

We remark that: all the behavior of the system is in the rules and no principle must be added for specify its transitions; this behavior can be deduced logically from the axioms; and finally, several variants can be defined only by changing the axioms PP.

8 Conclusion and Open Problems

In the previous sections we gave motivations and examples for the applicability of monoidal systems, aimed at showing their role in the formalization of locality, temporality, and multiplicity aspects. These aspects are crucial in biochemical contexts, where space, time, matter, and energy are involved in many specific forms: osmosis, polarity, gradient, channel, flow. Let us consider the oxidative phosphorylation, that is, the mitochondrial ATP-ADP cycle [1]. It is a fundamental life process associated to Krebbs' cycle that makes possible the production of ATP, the energetic chemical unity used in the biosynthesis processes (the passage $ATP \rightarrow ADP$ liberates energy). In the mitochondrial region the NADH molecule resulting from Krebbs' cycle transforms into $NAD^+ + H^+$ with an energy gain that makes possible, in several steps, and by means of specific molecules, the passage of the H^+ ions outside the mitochondrial membrane. According to the chemiosmotic theory, the protonic gradient, between the internal and external regions of the mitochondrial membrane, activates a ionic chanel that makes possible a flow of H^+ from the outside to the internal region and this flow inverts the reaction $ATP \rightarrow ADT + P$ into $ADP + P \rightarrow ATP$, and promotes, by using suitable vector molecules, many other passages across the membrane (e.g. the passage from outside of $NADH$, phosphorus and ADP).

Of course, the right manner to express formally all the aspects of this process depend strongly on the possibility of a good formal setting where specific aspects of molecules are adequately treated. We need a lot of work in order to obtain formal systems that can predict the behavior of biochemical processes, or only explain some of their features as consequences of axioms expressing general structural aspects, but certainly, discrete logical symbolic systems will help us to extract their pure informational aspect, giving some clue in the comprehension of the global logic which controls the enormous number of molecule manipulation systems cooperating in biological environments.

References

[1] Alberts B., Bray D., Lewis J., Raff M., Roberts K., Watson, J.D., Molecular Biology of the Cell, Garland Publishing Inc., New York, 1989.

[2] Dassow J., Păun G., Regulated Rewriting in Formal Language Theory, Springer-Verlag, Berlin, Heidelberg, 1989.

[3] Enderton H. B., A Mathematical Introduction to Logic, Academic Press, New York, 1972.

[4] Guarnieri F., Fliss M., Bancroft C., Making DNA add, Science, 273, 220-223, 1996.

[5] Head T., Formal language theory and DNA: an analysis of the generative capacity of recombinant behaviors, Bulletin of Mathematical Biology, 49(1987) pp. 737-759.

[6] Herman G. T., Rozenberg G., Developmental Systems and Languages, North-Holland, Amsterdam, 1975.

[7] Kari L., Rozenberg G., Salomaa A., L Systems, in [21], Vol. II, pp. 253-328, 1997.

[8] Lindenmayer A., Mathematical models for cellular interaction in development, I and II, J. Theoret. Biol., 18, pp. 280-315, 1968.

[9] Manca V., String Rewriting and Metabolism: A Logical Perspective, in: Computing with Bio-molecules. Theory and Experiments, Păun G. (ed.), Springer-Verlag, Singapore, pp. 36-60, 1998.

[10] Manca V, Martino M. D., From String Rewriting to Logical Metabolic Systems, in: G. Păun, A. Salomaa (eds.), Grammatical Models of Multi-agent Systems, Gordon and Breach Science Publishers, Topics in Computer Mathematics, Vol. 8, pp. 297-315, London, 1999.

[11] Manca V., C. Martin-Vide, G. Păun, Iterated GSM Mappings: A Collapsing Hierarchy, in: Jewels Are Forever, A. Salomaa, H. Mauer, G. Paun (eds.), pp.182-193, Springer-Verlag, New-York, 1999.

[12] Manca V., Logical String Rewriting, Theoretical Computer Science, Special Issue devoted to MFCS 98, 23rd International Symposium on Mathematical Foundations of Computer Science, 2001 (to appear).

[13] Manca V., Logical Representation of Grammatical Systems, International Workshop Grammar Systems 2000, July 3-7, 2000, Bad Ischl, Austria, to appear.

[14] Marcus S., Linguistic Structures and Generative Devices in Molecular Genetics, in: Cahier de Linguistique Théorique et Appliquée, Vol. 11, n. 2, pp. 77-104, 1974.

[15] Marcus S., Language at the Crossroad of Computation and Biology, in: Computing with Bio-molecules. Theory and Experiments, Păun G. (ed.), Springer-Verlag, Singapore, pp. 1-35, 1998.

[16] Păun G., Computing with membranes, *TUCS Research Report N.* 208, November 1998 (hhtp://www.tucs.fi).

[17] Păun G., Computing with membranes. An introduction, Bulletin of the EATCS 67, pp. 139-152, Febr. 1999.

[18] Păun G., P Systems with Active Membranes: Attacking NP Complete Problems, CDMTCS Technical Report 102, May 1999 (hhtp://www.cs.auckland.ac.nz/CDMTCS).

[19] Pawlak Z., Gramatyka i Matematika, Panstwowe Zakadi Wydawnietw Szkolnych, Warszawa, 1965.

[20] B. Rovan, A framework for studying grammars, *Proc. MFCS 81, Lect. Notes in Computer Sci.* 118, Springer-Verlag, 473–482, 1981.

[21] Rozenberg G., Salomaa A.(eds.), Handbook of Language Theory, 3 Voll., Springer-Verlag, Berlin, Heidelberg, 1997.

[22] Salomaa A., Formal Languages, Academic Press, New York, 1973.

[23] Smoryński C., Logical Number Theory, Springer-Verlag, Berlin, Heidelberg, 1991.

[24] Smullyan R. M., Theory of Formal Systems, Princeton Univ. Press, Princeton, New Jersey, 1961.

[25] D. Wood, Iterated a-NGSM maps and Γ-systems, *Inform. Control*, 32, 1–26, 1976.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 191 - 192.

Bags And Beyond Them

Solomon Marcus
Romanian Academy,Mathematics
Calea Victoriei 125, Bucuresti, Romania
e-mail: smarcus@stoilow.imar.ro

1 Introduction

I will use the word "bag" for what is usually understood by a multiset: a set A whose each element x exists in a number c(x) of copies of x.We will accept the general situation when A is of an arbitrary cardinal and c(x) too is an arbitrary cardinal. As a matter of fact, even if A is finite, but very (too) large and of a cardinal which is not exactly known (this is what happens in many situations occurring in physics, chemistry or biology) it is convenient to approximate A by an infinite set.

2 Bags as equivalence relations

Denote by E(A) the set of all copies of elements in A. The relation asserting that y belongs to c(x) is an equivalence relation in E(A), provided that we accept what is usually implicitly understood: if u, v, w are elements in E(A), then u is a copy of u; if u is a copy of v, then v is a copy of u; if u is a copy of v and v is a copy of w, then u is a copy of w. We can proceed in the opposite way too: Given a set A and an equivalence relation r in A, we can consider u as a copy of v exactly when urv; again any element is a copy of itself; if u is a copy of v, then v is a copy of u; a copy of a copy of u is still a copy of u.

3 What happens in some specific situations

It happened to me to meet some important bags in the field of linguistics, where the passing from etic to emic units (in phonology, in morphemics and in semantics) is just the move from individual elements to their equivalence classes, i.e., from copies to classes of copies. For instance, in phonology it was generally assumed that a phoneme is an equivalence class of sounds; each sound is a copy of any other sound in the same equivalence class, i.e., belonging to the same phoneme. However, in a more rigorous approach, as it was proposed by Kanger (1964) and improved by Marcus (1965, 1967), starting from the viewpoint proposed in descriptive linguistics (Harris 1961), we are faced in the mathematical modeling of the phoneme with a binary relation called "variation in the broad sense", which corresponds to what is called in the linguistic analysis of the phoneme "free variation". As it is proved in Marcus

(1967: 63), the binary relation "u and v are in the relation of variation in the broad sense" is not transitive, despite the fact that u and v belong to the same phoneme. So, if at a first glance the set of phonemes is a bag, where each sound is a copy of another one iff they belong to the same phoneme, in the more rigorous approach mentioned above the respective binary relation is no longer transitive, it is only reflexive and symmetric, i.e., what is called a tolerance relation. A similar situation occurs in the study of synonymy. At a first glance, the relation of synonymy is an equivalence relation and leads to the structure of a bag: two strings are synonymous if they have the same meaning (Marcus 1973, chapter IV). However, synonymy in natural languages is generally not transitive, because it is context dependent, as it can be seen on the example of "great", "large" and "big", each of them selecting specific contexts.

4 Tolerance spaces as an extension of bags

We reach in this way the idea to extend the notion of a bag, by considering the relation "u is a copy of v" as a tolerance relation, i.e., a binary relation which is reflexive and transitive (Pogonowski 1981, Shreider 1975). The associated topology is no longer the usual one; it is what is called a Cech topology (Cech 1966: chapter 3), which differs from the classical topology by the fact that the closure operator is replaced by a more general one, where the closure of the closure of a set A contains A, but it is not always equal to A. We have already used it in respect to learning processes (Marcus 1989, 1994).

References

E. Cech 1966 Topological Spaces. Prague: Publishing House of the Czech Academy.
Z.S. Harris 1961 Structural Linguistics. Chicago: University Press.
S. Kanger 1964 The notion of a phoneme. Statistical Methods in Linguistics 3, 43-48 (Stockholm).
S. Marcus 1965 Sur un ouvrage de Stig Kanger concernant le phoneme. Statistical Methods in Linguistics 4, 43-48.
S. Marcus 1967 Introduction Mathematique a la Linguistique Structurale. Paris: Dunod.
S. Marcus 1973 Mathematische Poetik. Frankfurt/Main: Athenaum.
S. Marcus 1989 Interplay of innate and acquired in some mathematical models of language learning processes. Revue Roumaine de Linguistique 34, 101-116.
S. Marcus 1994 Tolerance rough sets, Cech topologies, learning processes. Bull. Polish Academyof Science, Technical Science 42, 3, 471-487.
J. Pogonowski, Tolerance Spaces with Applications in Linguistics. Poznan: University Press. Yu. A. Shreider 1975 Equality, Resemblance and Order. Moscow: Mir.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 203 - 217.

Multiset and K-subset transforming systems

Taishin Yasunobu Nishida[*]
Faculty of Engineering, Toyama Prefectural University,
Kosugi-machi, 939-0398 Toyama, Japan

Abstract

We introduce K-subset transforming systems as a generalization of multiset transformation. A K-subset, which is a generalization of a multiset of which "multiplicities" is take values of a semiring, is considered by S. Eilenberg. We construct an example of K-subset transforming system which models a chaotic discrete dynamical system. We show that for every basic reaction of multiset transformation we can construct a K-subset transforming system which expresses the multiset transformation. We also show that for every phrase structure grammar there is a K-subset transforming system such that the system simulates derivations of the grammar.

1 Introduction

Recently a number of new computing models are proposed, quantum computing [15], DNA computing [10], membrane computing or P system [3, 9, 11, 12, 13], and so on. These new models and the "traditional" models, such as Turing machine, phrase structure grammar, term rewriting system, cellular automata, etc, give us quite different appearance. But we can find a common feature of them: they all obey discrete time development. We can say that computational science is a science of discrete dynamical systems, by contrast the physical science have been described by continuous differential equations.

In this paper we try to build a general framework of discrete dynamical systems. We adopt K-subset [5] to express objects in discrete dynamical systems. A K-subset is a generalization of a multiset or a multiset of which "multiplicities" take values of a semiring. Among many varieties of multiset theory, K-subset has a firm theoretical background [4]. Our model has a set of rules which are pairs of a condition and an action over K-subsets. Adding an initial K-subset, we obtain a K-subset transforming system.

Multiset transformation is a good model of chemical reaction. But in such a situation that a molecule of a protein changes its conformation according to a concentration of other molecules, such as pH or calcium ion, rational or real "multiplicities" will be useful. And in quantum computing, the "multiplicities" of quantum states are complex numbers. This is why we introduce non-integral multiplicities.

[*]Email:nishida@pu-toyama.ac.jp, URL:http://www.comp.pu-toyama.ac.jp/~nishida

After preliminaries describing semirings and K-subsets, we define K-subset transforming systems and give an example which models a chaotic discrete dynamical system in Section 3. Then we mention the relation between K-subset transforming systems and multiset transformation in Section 4. Finally, in Section 5, we show that K-subset transforming systems include string rewriting systems: phrase structure grammars and L systems. Although some results in this paper are obtained by only simulating existing models with K-subset transforming systems, we believe that K-subset transforming systems open up new vistas of computational science. Example 2 and Theorem 6 suggest the wide possibilities of K-subset transforming systems.

2 Preliminaries

First we introduce the notion of semiring from [5]. A set K is said to be a *semiring* if K has two operations addition and multiplication, K is a commutative monoid with respect to addition, K is a monoid with respect to multiplication, and addition and multiplication are connected by the following equations

$$\begin{aligned} x(y+z) &= xy+xz \\ (x+y)z &= xz+yz \end{aligned}$$

$$x0 = 0 = 0x$$

for every $x, y, z \in K$ where 0 is the unit element with respect to addition, $+$ stands for addition, and omitted $\cdot$ stands for multiplication. The unit element with respect to multiplication is denoted by 1. Thus for every element $x, y, z \in K$ we have

$$\begin{aligned} x+y &= y+x \\ x+(y+z) &= (x+y)+z \\ x+0 &= x \\ x(yz) &= (xy)z \end{aligned}$$

$$1x = x = x1$$

Clearly any ring is a semiring. We give a few examples of semiring which will appear in this paper.

Example 1 *The following sets are all semirings.*

1. $\mathcal{B} = \{0, 1\}$ *with the operations:*

$$0+0=0,\ 0+1=1+0=1,\ 1+1=1$$

$$00=01=10=0,\ 11=1.$$

So 0 is the unit element for addition and 1 is the unit element for multiplication. Notice that $\mathcal{B}$ is different from GF(2)[1].

[1]GF(2) is the only field of two elements and satisfies $1+1=0$.

2. *The set of all nonnegative integers* $\mathbb{N}$.

3. *The set of all real numbers* $\mathbb{R}$.

A semiring K is called *commutative* if for every $x, y \in K$ we have $xy = yx$.

Then we define K-subsets [5]. We assume that K is a nontrivial semiring, i.e., $0 \neq 1$, or equivalently K has at least two elements. We also assume that K is commutative. Let X be a set. A *K-subset* A of X is a function

$$A : X \to K.$$

For every $x \in X$ the element $A(x)$ of K is called the *multiplicity with which x belongs to A* or *multiplicity of x in A*. The union $\cup$ and the intersection $\cap$ of two K-subsets A and B of X are defined by

$$(A \cup B)(x) = A(x) + B(x) \text{ and}$$

$$(A \cap B)(x) = A(x)B(x).$$

For every $k \in K$ the operation kA is defined by

$$(kA)(x) = kA(x).$$

We note that every $\mathcal{B}$-subset B of X corresponds a normal subset S of X by the next equation.

$$S = \{x \in X \mid B(x) = 1\}.$$

A "normal" multiset is a collection in which elements may be duplicated, for example,

$$\{a, a, a, b, c, c\}.$$

The number of times an element occurs in a multiset is called its multiplicity. In this paper we denote a multiset in the form

$$\{\text{multiplicity of } a \cdot a, \text{multiplicity of } b \cdot b, \ldots\},$$

so the multiset above is expressed as

$$\{3 \cdot a, 1 \cdot b, 2 \cdot c\}.$$

Then we have the next proposition.

Proposition 1 *For a set X, there exist functions μ and ν such that μ maps every $\mathbb{N}$-subset A of X to a multiset $\mu(A)$ over X, ν maps every multiset M over X to an $\mathbb{N}$-subset $\nu(M)$ of X, and $\mu\nu$ and $\nu\mu$ are identities.*

Proof. The function μ and ν are defined by

$$\mu(A) = \bigcup_{x \in X \wedge A(x) > 0} \{A(x) \cdot x\}$$

and

$$\nu(M)(x) = \begin{cases} \text{the multiplicity of } x \text{ in } M & \text{if } x \in M \\ 0 & \text{otherwise} \end{cases}.$$

Then the conclusions are obvious. □

For every $x, y \in \mathbb{N}$ we define $x \dot{-} y$ by

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}.$$

Unless otherwise stated, for every predicate P, we assume that $\forall i\,(P)$ and $\exists i\,(P)$ stand for $\forall i \in \mathbb{N}(P)$ and $\exists i \in \mathbb{N}(P)$, respectively.

3 Definitions of K-subset transforming systems

Here we give the definition of a K-subset transforming system.

Definition 1 *A K-*subset transforming system *is a 4-tuple $G = \langle X, K, R, A_0 \rangle$ where X is a set, K is a semiring, R is a set of rules of the form*

$$\langle\text{condition}\rangle : \langle\text{action}\rangle$$

in which condition *is a closed predicate whose variables take values over X and K-subset of X and* action *consists a set of formulas that give a new K-subset from the current K-subset. If condition and/or action have infinitely many formulas, then rules may be expressed by a schema of rules. Usually, we omit, from actions, the definition of multiplicities of elements of X in the new K-subset which take the same multiplicities in the current K-subset. And $A_0 : X \to K$ is the initial K-subset.*

A K-subset A' is derived from a K-subset A by G if there is a rule whose condition is true for A and A' is obtained from A by the action of the rule. A K-subset transforming system G generates a sequence of K-subsets $(A_0, A_1, \ldots)$ in which A_n is derived from A_{n-1} by G for $n = 1, 2, \ldots$. If there is no rule whose condition is true for a K-subset A_n, then G derives no K-subset from A_n and the sequence is terminated at A_n.

Example 2 *Let $G = \langle \{y\}, \mathbb{R}, R, A_0 \rangle$ be an $\mathbb{R}$-subset transforming system where R consists of*

$$A(y) \geq 0 \;:\; A'(y) = -2A(y) + 1$$

$$A(y) < 0 \;:\; A'(y) = 2A(y) + 1$$

and $A_0(y) = x_0$ for some $x_0 \in [-1, 1]$. Then the multiplicities of y in the sequence $(A_0, \ldots, A_n, \ldots)$ generated by G give the trajectory of the discrete dynamical system

$$x_{n+1} = -2|x_n| + 1,$$

i.e.,

$$x_n = A_n(y).$$

This is an example of chaotic dynamical systems (Example 6.2.1 of [8]).

4 Multiset transformation and $\mathbb{N}$-subset transforming system

The $\mathbb{N}$-subset transforming system will be equal to the multiset transformation. For example, the sort program written in GAMMA [1, 2, 6] looks like:

Example 3 *Let $G = \langle X, \mathbb{N}, R, A_0 \rangle$ be an $\mathbb{N}$-subset transforming system where*

$$X = \{(1, x_1), \ldots, (n, x_n)\},\ x_i \in \mathbb{R},\ 1, \ldots, n \in \mathbb{N},$$

A_0 is an $\mathbb{N}$-subset of X, and R consists of the rule schema

$$\exists i \exists j\, (\exists x, y \in \mathbb{R})(A((i,x))A((j,y)) > 0 \wedge i < j \wedge x > y)\ :$$

$$A'((i,x)) = A((i,x)) \dot{-} 1,\ A'((j,y)) = A((j,y)) \dot{-} 1,$$
$$A'((i,y)) = A((i,y)) + 1,\ A'((j,x)) = A((j,x)) + 1.$$

Now obviously G generates the finite sequence $(A_0, \ldots, A_k)$ such that $i \leq j$ and $x \leq y$ if $A_k((i,x))A_k((j,y)) > 0$.

Example 3 is generalized to the basic reaction of GAMMA program.

Theorem 2 *Let X be a set and let*

$$G : x_1, \ldots, x_n \to A(x_1, \ldots, x_n) \Leftarrow R(x_1, \ldots, x_n)$$

be a basic reaction of multiset over X where $x_1, \ldots, x_n$ are variables and R and A are of arity n [6]. Then there is an $\mathbb{N}$-subset transforming system H such that G transform a multiset M to M' if and only if H generates $\mathbb{N}$-subset M' from M.

Proof. Let $H = \langle X, \mathbb{N}, P, A_0 \rangle$ be an $\mathbb{N}$-subset transforming system where P consists of

$$\begin{aligned}(\exists x_1, \ldots, x_n \in X) R(x_1, \ldots, x_n) \quad : \quad & B'(x_i) = B(x_i) + \nu(A(x_1, \ldots, x_n))(x_i) \dot{-} 1, \\ & \qquad i = 1, \ldots, n \\ & B'(y) = B(y) + \nu(A(x_1, \ldots, x_n))(y) \\ & \qquad \text{for } y \notin \{x_1, \ldots, x_n\}\end{aligned}$$

and $A_0 = \nu(M_0)$ where M_0 is the initial multiset for G and ν is the function defined in Proposition 1. Then the conclusion follows immediately. □

We do not treat sequential and parallel composition operators of GAMMA [6]. But by Theorem 4 in Section 5, K-subset transforming systems have computational universality.

5 Phrase structure grammars, L systems, and K-subset transforming systems

In this section we consider the relation between string rewriting systems and K-subset transforming systems. We assume the reader is familiar with basics of phrase structure grammars (see [14]) and L systems (see [7]).

Let Σ be a finite alphabet. A $\mathcal{B}$-subset A of $\mathbb{N} \times \Sigma$ is said to be linearizable if A satisfies

1. For every $i \in \mathbb{N}$ and $a, b \in \Sigma$ such that $a \neq b$ we have $A((i,a))A((i,b)) = 0$.
2. For every $i < j < k$ and every $a, b, c \in \Sigma$ we have $\neg(A((i,a)) = 1 \wedge A((j,b)) = 0 \wedge A((k,c)) = 1)$.

For a linearizable $\mathcal{B}$ subset A, a mapping $\phi : A \to \Sigma^*$ or Σ^ω is defined by

$$\phi(A) = \begin{cases} a_i \cdots a_j \in \Sigma^* & \text{if } A((i,a_i)) = \cdots = A((j,a_j)) = 1 \wedge \\ & A((k,b)) = 0 \text{ for } k < i, k > j \\ a_i \cdots a_j \cdots \in \Sigma^\omega & \text{if } A((j,a_j)) = 1 \text{ for some } i \in \mathbb{N} \text{ and every } j \geq i \end{cases}.$$

Then the next theorem describes that $\mathcal{B}$-subset transforming systems include phrase structure grammars.

Theorem 3 *Let $G = \langle V, \Sigma, P, S\rangle$ be a grammar. Then there exists a $\mathcal{B}$-subset transforming system H such that for every sentential form w generated by G there is a $\mathcal{B}$-subset A of $\mathbb{N} \times V$ generated by H which satisfies*

$$w = \phi(A).$$

Proof. Let $H = \langle \mathbb{N} \times V, \mathcal{B}, R, A_0\rangle$ where $A_0((1,S)) = 1$, $A_0((i,x)) = 0$ for $i \neq 1$ and $x \in V$, and R consists of the following rule schema:
For every $a_1 \cdots a_k \to b_1 \cdots b_l \in P$ and $l \geq 1$

$$\exists i\, (A((i,a_1)) = \cdots = A((i+k,a_k)) = 1 \wedge a_1 \cdots a_k \to b_1 \cdots b_l) :$$
$$A'((i,b_i)) = \cdots = A'((i+l,b_l)) = 1,$$
$$A'((i,a_1)) = \cdots = A'((i+k,a_k)) = 0,$$
$$A'((i+l+j, c_{i+k+j})) = A((i+k+j, c_{i+k+j})) \text{ for } j > l, \text{ and}$$
$$A'((j,c_j)) = A((j,c_j)) \text{ for } j < i.$$

For every $a_1 \cdots a_k \to \varepsilon \in P$

$$\exists i\, (A(i,a_1)) = \cdots = A((i+k,a_k)) = 1 \wedge a_1 \cdots a_k \to \varepsilon) :$$
$$A'((i,a_i)) = \cdots = A'((i+k,a_k)) = 0,$$
$$A'((i+j-1, c_{i+k+j})) = A((i+k+j, c_{i+k+j})) \text{ for } j > 0, \text{ and}$$
$$A'((j,c_j)) = A((j,c_j)) \text{ for } j < i.$$

First we observe that every $\mathcal{B}$-subset A of $\mathbb{N} \times V$ generated by H is linearizable. Then the definition of ϕ leads the conclusion. □

Since the above theorem says that K-subset transforming systems can simulate type 0 grammars, we have the following theorem.

Theorem 4 *The K-subset transforming systems generate all recursively enumerable languages. There is a K-subset transforming system generating a sequence of K-subsets which is not recursively enumerable.*

Proof. The first assertion is a corollary of Theorem 3. Since a chaotic dynamical system shows quite different behaviour by any infinitesimal change in the initial value, the different initial $\mathbb{R}$-subset in Example 2 gives the different $\mathbb{R}$-subset transforming system. So the cardinality of possible $\mathbb{R}$-subset transforming systems in Example 2 is the cardinality of continuum. Then the second assertion is true. $\square$
We note that, by Charch's hypothesis, the class of effectively computable K-subset transforming systems must coincide with the class of Turing machines.

Next we consider L systems.

Theorem 5 *Let $G = \langle \Sigma, P, \#, w\rangle$ be a $(1,1)L$ system where $\#$ is the environmental marker not in Σ. Then there is a $\mathcal{B}$-subset transforming system H such that for every $u \in \Sigma^+$ derived by G, H generates the linearizable $\mathcal{B}$-subset A of $\mathbb{N} \times (\Sigma \cup \{\#\})$ satisfying*

$$u\# = \phi(A).$$

Proof. Let $H = \langle X, \mathcal{B}, R, A_0\rangle$ be the $\mathcal{B}$-subset transforming system where

$$X = \{-1\} \cup \mathbb{N} \cup \{\$\} \times \mathbb{N} \cup \mathbb{N} \times (\Sigma \cup \{\#\}) \cup (\{0,1\} \times \mathbb{N}) \times (\Sigma \cup \{\#\}),$$

$$A_0((i, a_i)) = 1,\ A_0((l+1, \#)) = 1,\ \text{and } A_0(x) = 0 \text{ for other } x \in X$$

where $w = a_0 \cdots a_l$, and R has the following rules:

1. $\exists i \exists a \in \Sigma \cup \{\#\}\, (A((i,a)) = 1) : A'(((0,i),a)) = 1,\ A'((i,a)) = 0.$

2. $\forall i \forall a \in \Sigma \cup \{\#\}\, (A((i,a)) = 0) \wedge \exists a, x \in \Sigma(A(((0,0),a))A(((0,1),x)) = 1) \wedge$
$$(\#, a, x) \to b_1 \cdots b_k \in P :$$
$$A'(((1,0),b_1)) = \cdots = A'(((1,k-1),b_k)) = 1,$$
$$A'(((0,0),a)) = 0, A'(1) = 1, A'((\$, k-1)) = 1.$$

2'. $\forall i \forall a \in \Sigma \cup \{\#\}\, (A((i,a)) = 0) \wedge \exists a \in \Sigma(A(((0,0),a))A(((0,1),\#)) = 1) \wedge$
$$(\#, a, \#) \to b_1 \cdots b_k \in P :$$
$$A'(((1,0),b_1)) = \cdots = A'(((1,k-1),b_k)) = 1,$$
$$A'(((0,0),a)) = 0, A'(-1) = 1, A'(((1,k),\#) = 1.$$

3. $\exists j\, (A(j) = 1) \wedge \exists l\, (A((\$, l)) = 1) \wedge$
$$\exists a, x, y \in \Sigma(A(((0,j-1),x))A(((0,j),a))A(((0,j+1),y)) = 1) \wedge$$
$$(x, a, y) \to b_1 \cdots b_k \in P :$$
$$A'(((1,l+1),b_1)) = \cdots = A'(((1,l+k),b_k)) = 1, A'(((0,j),a)) = 0,$$
$$A'(j+1) = 1, A'(j) = 0, A'((\$, l)) = 0, A'((\$, l+k)) = 1.$$

4. $\exists j\,(A(j)=1)\wedge\exists l\,(A((\$,l))=1)\wedge$

$$\exists a,x\in\Sigma(A(((0,j-1),x))A(((0,j),a))A(((0,j+1),\#))=1)\wedge$$

$$(x,a,\#)\to b_1\cdots b_k\in P:$$

$$A'(((1,l+1),b_1))=\cdots=A'(((1,l+k),b_k))=A'(((1,l+k+1),\#))=1,$$

$$A'(((0,j),a))=0,A'(j)=0,A'((\$,l))=0,A'(-1)=1.$$

5. $A(-1)=1\wedge\exists i\exists a\in\Sigma\cup\{\#\}(A(((1,i),a))=1):$

$$A'((i,a))=1,A'(((1,i),a))=0.$$

6. $A(-1)=1\wedge\forall i\forall a\in\Sigma\cup\{\#\}(A(((1,i),a))=0):A'(-1)=0.$

Now we show that for every $\mathcal{B}$-subset A of $\mathbb{N}\times(\Sigma\cup\{\#\})$ satisfying $u\#=\phi(A)$ for some $u\in\Sigma^+$, $u\Rightarrow_G v$ if and only if there exists a $\mathcal{B}$-subset B which is generated from A by H and $\phi(B)=v\#$. Let $u=a_0\cdots a_{n-1}$ where $a_i\in\Sigma$, $i=0,\dots,n-1$, let $A((0,a_0))=\cdots=A((n-1,a_{n-1}))=A((n,\#))=1$, and let $A((j,a))=0$ for $j<0$ or $j>n$. Then by iterating the rule 1 $n+1$ times, we have $\mathcal{B}$-subset A_1 such that

$$A_1(((0,0),a_0))=\cdots=A_1(((0,n-1),a_{n-1}))=A_1(((0,n),\#)=1.$$

We note that the rule 2 cannot be iterated until the rule 1 is iterated $n+1$ times by the first condition of the rule 2:

$$\forall i\forall a\in\Sigma\cup\{\#\}\,(A((i,a))=0).$$

Next rules 2, 3, and 4 (or 2') simulate the derivation of G from left to right. After the rule 4 (or 2') is iterated, we have $\mathcal{B}$-subset A_2 satisfying

$$A_2(((1,0),b_0))=\cdots=A_2(((1,m),b_m))=A_2(((1,m+1),\#))=1,$$

$$A_2(-1),\text{ and }b_0\cdots b_m=v.$$

Finally rules 5 and 6 generate the desired $\mathcal{B}$-subset B. Then it is proved that H generates $\mathcal{B}$-subset A of $\mathbb{N}\times(\Sigma\cup\{\#\})$ if only if G generates $u\in\Sigma^+$ such that

$$u\#=\phi(A).$$

If G generates ε, then H generates $\mathcal{B}$-subset A_ε and A_ε derives nothing where

$$A_\varepsilon(((0,0),\#))=1,\ A_\varepsilon(x)=0\text{ for other }x\in X.$$

Since all L systems generate ε from ε, this makes no problem. □

The $\mathcal{B}$-subset transforming system constructed in the above proof is quite unefficient. It simulates one step derivation of an L system with many steps. We should find K-subset transforming systems which can generate strings in parallel.

But by considering an $\mathbb{N}$-subset of $\Sigma^*\times\mathbb{N}$, we can measure the multiplicity of a word, that is, the total number of different derivations of a word in an L system.

Theorem 6 *Let $G = \langle \Sigma, P, \#, w \rangle$ be an L system. Then there is an $\mathbb{N}$-subset transforming system H such that for every word u derived by G in i steps H generates an $\mathbb{N}$-subset A of $\Sigma^* \times \mathbb{N}$ and that $A((u,i))$ gives the multiplicity of u.*

Proof. Let $H = \langle \Sigma^* \times \mathbb{N}, \mathbb{N}, R, A_0 \rangle$ where $A_0((w,0)) = 1$, $A_0((x,i)) = 0$ for $x \neq w$ or $i \neq 0$ and R consists of the following rule schema

$$\exists i \exists u \in \Sigma^* (A((u,i)) > 0 \wedge \forall v \in \Sigma^* (u \Rightarrow_G v)) :$$

$$A'((v, i+1)) = A((v, i+1)) + A((u,i)), A'((u,i)) = 0.$$

Then obviously $A((u,i))$ gives the multiplicity of u derived by G in i steps. □

References

[1] J. Banâtre, A. Coutant, and D. Le Metayer, A parallel machine for multiset transformation and its programming style, *Future Generations Computer Systems* **4** (1988) 133–144.

[2] J. Banâtre and D. Le Métayer, Programming by multiset transformation, *Communications of the ACM* **36** (1993) 98–111.

[3] J. Dassow and G. Păun, On the power of membrane computing, *Journal of Universal Computer Science* **5** (1999) 33–49.

[4] W. D. Blizard, The development of multiset theory, *Modern Logic* **1** (1991) 319–352[2].

[5] S. Eilenberg, *Automata, Languages, and Machines Volume A*, (Academic Press, New York, 1974).

[6] C. Hankin, D. Le Métayer, and D. Sands, Refining multiset transformers, *Theoretical Computer Science* **192** (1998) 233–258.

[7] G. T. Herman and G. Rozenberg, *Developmental Systems and Languages* (North-Holland, Amsterdam, 1975).

[8] M. Martelli, *Discrete Dynamical Systems and Chaos*, (Longman Scientific & Technical, Harlow, 1992).

[9] G. Păun, Computing with membranes, *Journal of Computer and System Sciences*, to appear, (and *Turku Centre for Computer Science-TUCS Report* No 208, 1998 (http://www.tucs.fi)).

[2]There is a correction to this paper. But you need not look at the correction. The correct correction is
"Item [8] on p. 349 of this paper should have read as follows:
[8] Blizard, W., Dedekind Multisets and Function Shells, *Theoretical Computer Science* **110** (1993) 79–98."

[10] G. Păun, G. Rozenberg, and A. Salomaa, *DNA Computing*, (Springer, Berlin, 1998).

[11] G. Păun, Computing with membranes. An introduction, *Bulletin of the EATCS* **67** (1999) 139–152.

[12] G. Păun, Computing with membranes. A correction, two problems, and some bibliographical remarks, *Bulletin of the EATCS* **68** (1999) 141–144.

[13] G. Păun, P systems: an early survey, *The Third International Colloquium on Words, Languages and Combinatorics* March 2000, Kyoto (Proceedings will be published by World Scientific, Singapore).

[14] A. Salomaa, *Formal Languages*, (Academic Press, New York 1973).

[15] P. W. Shor, Algorithm for quantum computation: discrete log and factoring, in: *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science* (1994).

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 203 - 217.

Computing with Membranes (P Systems): Twenty Six Research Topics[1]

Gheorghe PĂUN

Institute of Mathematics of the Romanian Academy
PO Box 1 – 764, 70700 Bucureşti, Romania
E-mail: `gpaun@imar.ro`

Abstract. The aim of these notes is to state a series of open problems and, mainly, research topics about P systems. They can be clustered in three classes: questions dealing with "classic" topics in automata and language theory, questions motivated by the possible usefulness of P systems as computing models (implementation and complexity issues), and questions related to the fields where the P systems are inspired from, biology and biochemistry. Precise open problems can be found practically in all papers published or distributed so far on the web; here we are mainly interested in research directions, in classes of problems.

A Wealth of Research Topics

The reader is supposed to already be familiar with P systems, basic variants and basic results included, so I do not recall definitions, proofs, and theorems in a formal manner. The current bibliography of the domain, given at the end of this discussion, can be helpful to this aim. In particular, Chapter 3 from the monograph [P2] is recommended, as the first systematic survey of the domain (however, at the level of October 1999, which is not irrelevant for P systems study: several of the papers mentioned in the bibliography are dated later).

I only recall the picture in Figure 1, illustrating the idea of a membrane structure, as well as a list of keywords, naming ingredients of P systems of various types: membrane, elementary membrane, skin membrane, membrane structure, label, region, outer region, object, symbol-object, string-object, multiset, evolution rule, communication, commands *here, in, out*, target, nondeterministic communication, concentration, electrical charge (polarization), dissolving a membrane (action δ), increasing the thickness of a membrane (action τ), configuration, transition, computation, halting, internal/external output, active membrane, dividing a membrane.

So, let us begin directly by discussing possible directions for research. I am warning about the fact that these research directions are not all of the same generality, difficulty and/or importance, moreover, they are not ordered according to any conceivable criterion, such as the generality, difficulty, and/or importance. In particular, it is possible that some questions are easy to settle, while others might be close to nonsense. What I claim is that these questions deserve some efforts to clarify them, starting with the very problem whether or not they are trivial and/or

[1]Research supported by the Direcció General de Recerca, Generalitat de Catalunya (PIV), and the Polytechnical University of Madrid.

irrelevant. In this moment, nobody has done this effort. (Needless to say that I would be very P-indebted to the reader for any feedback.)

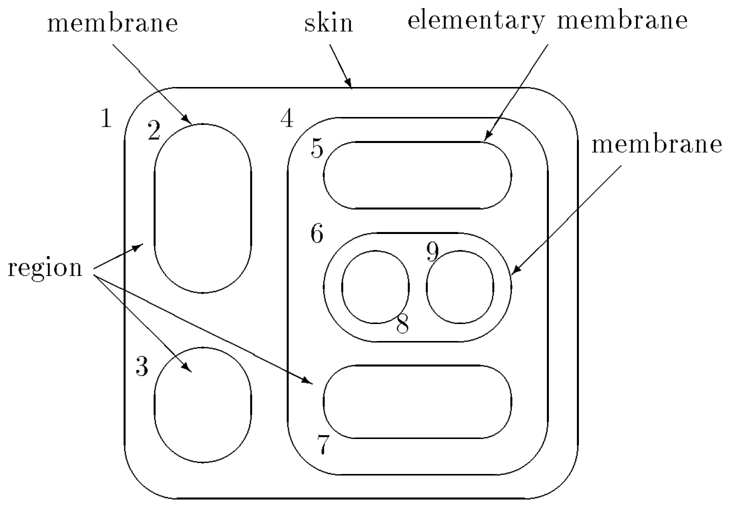


Figure 1: A membrane structure

a. Computing with membranes starts from the analogy of processes which take place in the complex structure of a living cell with a computing process. In the style of other branches of Natural Computing, we learn from (alive) nature new computing models and strategies (let us say, paradigms), but it is not clear in this moment whether or not we have to go back to biochemistry for implementing the new computing models (like in DNA Computing) or to the electronic computer (to the general purpose one, or to a specially designed one) for implementation (like in Neural Networks and Genetic Algorithms). Figure 2 illustrates this dilemma, which, in my opinion, is the most fundamental one for membrane computing in this moment.

b. We have also to be prepared for the case that no implementation of P systems will be done (that is, no implementation of a real practical interest), that the attempt will remain forever *in info*.

A sort of a metaquestion appears here: (at least up to now) the kind of problems and, mainly, the kind of proof tools addressed in P systems area are very similar to those in automata and language theory; however, in the basic membrane computing model, we do not deal with strings and languages, but with *multisets* of atomic *objects*. It is not the syntax what matters here, but the numerical vectors which characterize the multiplicity of copies of the objects from a given region. Then, P system theory (let me call it so) is a branch of what? Of *Formal Multiset Theory*, instead of *Formal Language Theory*? This sounds interesting, especially if we take into account that there are a series of papers devoted to multiset mathematics and manipulation (see, e.g., [1], [3], [22] and their references; remark the fact that in some papers one says "bag" instead of "multiset"). Then, P systems can be seen as a part of the *generative* theory of multisets (with the observation that they are already *distributed* systems, corresponding to grammar systems in language theory; the simpler, non-distributed case still waits for a systematic study (providing that a mathematical or a "practical" interest for such a study will be identified).

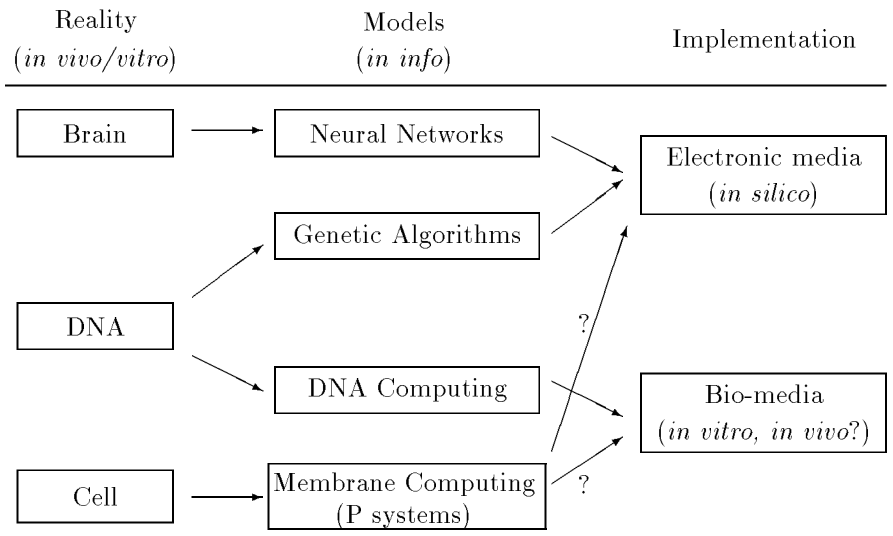


Figure 2: The four domains of Natural Computing

Actually, a really huge number of technical problems are in circulation in the P systems area and they already can motivate a Formal Multiset Theory. The whole program of formal language theory can be repeated here, irrespective of the fact whether or not we deal with languages or with multisets: generative/computing power; closure properties (by the way: what corresponds to an Abstract Family of Languages in terms of multisets? how does look an AFM = Abstract Family of Multisets, and the AFM theory?); necessary conditions and counterexamples (such tools are almost completely missing in this moment, but they are urgently necessary in order to settle such problems like finding infinite hierarchies and for separating the power of classes of P systems); decidability (classic – emptiness, finiteness, equivalence, etc – but also specific: is a specified membrane ever dissolved/divided?); comparison of classes of P systems among them and with other number or language generating devices (maybe taking the equality modulo Parikh images in the case of languages); the power of small systems (first, we have to define "small", hence to consider measures of descriptional complexity); and so on and so forth.

I do not persist in this direction, the reader can consult the bibliography of P systems, or can imagine him(her)self such problems.

c. Fundamental to P systems is not only the fact that (in the basic variant) we work with multisets, but also the *membrane structure*, with all consequences of that, especially the possibility of working in separate *regions*, arranged in a hierarchical fashion, and to *communicate* among regions. It is important to emphasize this: the membranes are *separators* and *channels of communication*. Having in mind the biological source of inspiration, we can imagine the membranes as physical objects, but this can be a serious limitation when looking for implementations. Any kind of virtual separators which allow (selective) communication among the delimited regions can play the role of membranes.

Continuing the previous ideas, computing with membranes should be interpreted in a very general manner, as a *framework* where distributed processes take place in a membrane structure (a hierarchical arrangement of regions), with an essential role played by communication. Keeping close to the biological source, P systems can be viewed as a general architecture of "living organisms", in the sense of Artificial Life (see, e.g., [13] and [21]). Whether or not the "life" of a system is to be interpreted as a computation or as a process with another meaning depends on the observer/user. The generality of the approach suggests to consider "incarnations" of P systems in other domains bearing some kind of "life" (in the sense of a development in time), such as logics and formal systems.

d. Regarding the central role of communication, the following question is of a clear mathematical and epistemological interest: what about the possibility of considering a class of P systems, meant to compute, where no rule for objects evolution appears, but only rules governing object communication from a region to another one. Because no object can be produced inside the system (there is no rule of the form $a \rightarrow bc$, thus increasing the number of occurrences of objects in the system), we have to consider rules for bringing objects from a specified source, for example, from the outer region. This can be done by either considering rules in the outer region of the form $b \rightarrow b_{in}, c \rightarrow c_{in}$, with the meaning that a copy of b and a copy of c are introduced in the skin membrane, or by considering a new command associated with objects, besides *here, in, out*, for instance *come*; rules introducing symbols b_{come} will be used in the skin membrane only, with the obvious meaning, of bringing a copy of b from outside the system.

Note that even simple rules of the form $a \rightarrow b$ are not allowed, hence we have either to allow changing of objects when passing through a membrane, or to simulate them, for instance, by sending the symbol a outside the system and, at the same time, bringing b from the outer region. If the rule $a \rightarrow b$ is to be used in a low level region, this does not look at all trivial.

The main mathematical difficulty stands in defining the communication among regions in such a way to get non-trivial computing results by using as elegant and as weak rules as possible. Some conditions should be observed in order to perform communication steps, for instance, in the form of predicates depending on the communicated objects and the contents of regions. If these predicated will be powerful, then the system will be powerful, but *too* powerful predicates will make the things non-interesting. (Maybe register machines can be used in order to prove results about such purely-communicative P systems.)

e. Let us stay close to biology. Several important laws and characteristics of what happens in biology and biochemistry (at the level of living cells) were practically ignored up to now when defining P systems. For instance, a fundamental law of chemistry is the *conservation of matter.* Rules of the form $a \rightarrow aa$ are creating a copy of a from nothing. *In info* this can be done, *in vivo* or *in vitro* it never happens... Can the conservation law be incorporated in the membrane computing area? An easy solution seems to be to consider that the system brings new objects from its environment (from the outer region), as suggested above for systems completely based on communication and without using evolution rules. Still, the problem is not trivial: if a piece of "raw material" a' is meant to become a together with one more copy of a, this means that we need a rule of the form $aa' \rightarrow aa$, which means cooperation. Cooperative systems are non-interestingly powerful. An elegant way to handle "raw materials" remains to be imagined.

Of course, it is much easier to handle "matter destroying rules", of the form $a \rightarrow \lambda$: instead of the empty string we can consider a dummy object, #, just denoting "garbage", which never evolves and which, if necessary, is sent out the system. (As usual in nature/life, the difficulty is to create, not to destroy...)

f. The trigger of defining P systems was the assertion, found in several places, that the processes taking place in a living cell, involving manipulations of chemical compounds, *energy*, and information, can be considered as computing processes. Up to now, almost exclusively chemical compound evolution was captured in P systems, by means of object evolution. What about energy?

I said "almost exclusively", because energy is implicitly present in the definition of certain ingredients. For instance, dissolving a membrane when using a certain rule, say, $a \rightarrow b\delta$, can be considered to correspond to destroying the membrane because the transformation $a \rightarrow b$ is accompanied by producing a large quantity of energy, which breaks the membrane.

Similarly, when defining a priority among rules, the interpretation can be again related to energy: rule $a \rightarrow b$ has priority on rule $c \rightarrow d$ because the reaction it models is more active than the reaction modeled by the latter rule; moreover, rule $c \rightarrow d$ cannot be applied at the same step with rule $a \rightarrow b$ just because the energy available was consumed by $a \rightarrow b$ (and not because the two rules compete for the same objects).

Thus: what about explicitly introducing the energy in the model?

In some sense, this can be easily done: consider a special object, e, as denoting a *quanta of energy*, and use it in the rules of a system as using any other object. For instance, a rule of the form $aee \rightarrow b$ means that object a is transformed into object b at the cost of two energy quanta; on contrary, $a \rightarrow bee$ means producing two energy quanta during transforming a into b. This again brings cooperation into system, of a well-restricted type.

A problem can arise about using or not rules of the form $a \rightarrow ee$, or even $e \rightarrow ab$. Can energy be converted into substance or conversely? How this can be done? (Which kind of $E = mc^2$-like formula we have to observe?)

g. The interpretation of the priority relation in the above "strong" sense reminds to us the use of an order relation in ordered grammars from the regulated rewriting area (see [11]): if $A \rightarrow u \ > \ B \rightarrow v$, then the rule $B \rightarrow v$ is not allowed to be used if the rule $A \rightarrow u$ can be used. What about an interpretation of the priority in a "weaker" way: a rule may be applied if there are objects to which no rule of a higher priority can be applied, irrespective of whether or not such rules of a higher priority can be applied to other objects.

An example can clarify the difference between the two interpretations. Consider that we have the rules $r_1 : ab \rightarrow cc$, $r_2 : b \rightarrow d$, with the priority $r_1 > r_2$, and the multiset represented by $aabbb$. The rule r_1 can be applied to the two pairs of objects ab. In the strong interpretation of the priority, the rule r_2 cannot be applied to the remaining copy of the object b, this b passes unchanged to the next configuration; consequently, the result is $ccccb$. In the weak interpretation, the rule r_2 can be applied at the same step with r_1 to the remaining copy of b, because r_1 cannot use it; the result is $ccccd$.

Up to now, only the strong interpretation has been investigated. The weak one only appears by a mistake in an example from the technical report version of [P15], also used in [P16] (see the correction from [P19]). What about P systems using a priority relation in the weak interpretation sense?

h. The conservation law is fundamental, but also the *reversibility* is basic to biochemistry. Many reactions are reversible, can go in either way (maybe depending on a change of reaction conditions, for instance, temperature – hence energy availability). In dynamic systems, reversibility is somewhat opposed to nondeterminism: we have to uniquely find the previous configuration of the system starting from the present configuration. Strictly chemically speaking, reversibility means reversing the direction of an equation. What about reversible P systems in the dynamic systems sense, what about "local reversible P systems", with the rules allowed to be used in both senses (for each $u \to v$ we have to also add $v \to u$ to the same region)? (By the way, it is known since many years, [2], that reversible Turing machines are computationally universal. Can reversibility be brought "for free" to P systems area from Turing machines area?)

i. At the border of biochemistry and mathematics we can find many further questions. One of them concerns the *determinism*. One of the central interesting features of computing with membranes is the inherent nondeterminism of P systems. This corresponds to what happens in cells and test tubes, where the chemical compounds swim in a solution, in a closed space, and evolve nondeterministically according to certain rules. If we attempt to implement membrane computing on the usual computer, then a big problem (= difficulty) appears: we have to simulate nondeterminism on a deterministic machine (still more: we have to simulate parallelism on a sequential machine). How to do this depends on the computer used and on the programming language used, and this is a practical problem. From a mathematical point of view, the problem is to consider deterministic P systems (what this precisely means, for instance, in the case of cooperating systems, is already a question) and to investigate their power. (The determinism versus nondeterminism question has a glorious history in automata theory and in Lindenmayer system theory, so suggestions can be found in these areas.)

j. A somehow related question: what about P systems with the same rules used in all regions? The use of rules associated with regions is motivated by the fact that rules model chemical reactions and these reactions need specific conditions, which can be assumed to be specific to regions. Various regions have different reaction conditions, hence different rules can act in each of them. Still, the chemistry is universal and the regions of the same cell should have a certain degree of uniformity. Which is the power of "uniform P systems", with the same set of rules acting in all regions?

In some sense, in P systems with active membranes we have such a case, but membranes themselves appear in the rules. The problem stated above is of interest for P systems without active membranes, in the sense of the paper [P15].

k. The previous question concerns, in fact, a *normal form* of P systems, which is a more general research topic. At least as important as the form of rules is the shape of the membrane structure. In specific circumstances, specific shapes could be more realistic than others, easier to implement than others. The number of different membrane structures grows fast with the number of membranes. In principle, we have to ask whether or not, given a P system with any given membrane structure, we can (effectively) construct an equivalent P system with another membrane structure (supplementary restriction: preserving the number of membranes).

Results of this type appear in [P22] and [P27]. For systems which are computationally complete, when this result is obtained by using a restricted number of membranes, the form of the system is already well restricted. Still, the question of finding normal forms (especially from the point of view of the shape of the membrane structure) requests further research efforts.

l. In living cells, the "objects" are moved from a region to another one, through membranes, by making use of the *protein channels* present in membranes, or because of electrical charges, or randomly, or, mainly, because of different concentrations in neighboring regions. Communication controlled by the concentration of objects was considered only in [P5], but its importance in biology motivates a more systematic investigation.

Actually, this is related to a more general topic, of defining the processes in a region depending on the *contents* of that region. For instance, the rules themselves could be chosen according to the objects present in a region. In real cells, division appear also depending on the contents of the cell, not necessarily entailed by a single object (sometimes, the cell divides just because it contains "too much" material, and it is necessary to create a supplementary space). A more extended dependency of evolution on region contents is natural to be considered.

m. We have already reached the "territory" of mathematical problems, having or not direct biochemical motivations and/or possible practical implications. A very fruitful question, of a rather common place in all computability areas, concerns *hierarchies*. Is $n+1$ strictly stronger than n? In general, it is expected to be so. Many parameters are intimate to the P systems structure, some of them of a classic form (corresponding to descriptional complexity measures in formal language theory, see, e.g., [12]), others are specific to the new devices. The number of catalysts, of membranes, of rules (in total or the maximal number of rules in a region) are of the first type, the depth of the membrane structure (the height of the tree associated with it), its width or "outdegree" (again, defining these notions for the associated tree) are of the second type. When characterizations of the power of Turing machines are obtained by systems of a given size (for instance, as the number of membranes), the hierarchies collapse, but when no such characterizations are known the problem whether or not a given parameter induces an infinite hierarchy remains to be investigated.

An important detail: because of the existence of universal Turing machines and of universal type-0 Chomsky grammars (see an explicit construction of such a grammar in [4]; small universal Turing machines can be found, e.g., in [18]), a proof that many hierarchies collapse can sometimes be found in a rather easy way. Namely, when a class of P systems is computationally complete and the proof of such a result starts from a Turing machine or a type-0 Chomsky grammar and it constructs an equivalent P system, then, by starting from a universal machine or grammar we get a universal P system, a *fixed* one, which can generate any given set M of numbers (as usual in P systems area) by changing the objects initially present in the system. The membrane structure and the rules will remain the same, only the objects in the initial configuration will depend on M. In this way, all parameters which deal with membranes and rules will remain unchanged. The size of the universal P system will be an upper bound for all hierarchies on parameters related to membranes and rules (but not on those related to the initial objects present in the system). Finding the smallest value of these parameters, sufficient for obtaining computational completeness, is another question, expected in many cases to be of a serious difficulty.

It appears here a really classic question. Many proofs of the computational completeness of P systems of various types start from matrix grammars with appearance checking (known to characterize the recursively enumerable languages) in the binary normal form (see Lemma 1.3.7 in [11]). However, no universal grammar of this type is known, so, we cannot get collapsing hierarchies in the way discussed above. Finding a universal matrix grammar, in a sort of binary normal form (with all non-initial matrices containing only two rules) is, therefore, a formal language theory problem of a definite interest for P system theory.

n. I have several times mentioned the possibility of describing the membrane structure of a P system by means of a tree. This has been already pointed out in [P15] and [P21], and considered again in [P22] as a starting point of a generalization of the idea of a membrane structure, to supports of P system-like devices working on graphs different from trees (only planar graphs were considered in [P22]). Here stands a more general question, which is actually a two-fold one: (1) to make use of this mathematically nice description of a membrane structure by a tree and to work on trees in a systematic way, and (2) to consider systematically the idea of working on graphs of more general forms. Of course, the biochemical intuition of a membrane structure is lost when discussing about trees (dissolving a membrane means to remove a node from the tree and to link all direct descendants of the node to the parent node, communicating objects through membranes means to move objects up and down in the tree, etc), but, maybe, tools and suggestions from graph theory will come into the stage.

o. Another large class of problems appears when considering string-objects instead of symbol-objects. This case was already investigated in [P15], [P9], [P14], [P24], and recently in [P31] and [P32], but a systematic study is still missing. When dealing with strings we need string processing rules; rewriting rules and splicing rules were considered, other types of rules not (such as insertion-deletion or context addition, as common in the area of contextual grammars, see [16], operations inspired from the genetic area as those considered in [10], and so on). "Purely biological P systems", involving operations inspired from biology, look of interest (at least aesthetically). What is clear is that string processing plus distributed computing in the sense of P systems is very powerful, "small" and "simple" P systems are to be expected to characterize the recursively enumerable languages. As usual when inventing a new class of P systems, also when looking for string processing P systems we have to look for systems which are as elegant as possible, using as simple as possible ingredients.

A somewhat strange idea concerning the communication of string-objects from a region to another one is the following, leading to a sort of "P systems based on worms processing": when communicating strings from one region to another one, through a membrane, proceed step-by-step (symbol-by-symbol?), in such a way that a string can have a part in one region and another part in another region (maybe, a string can have parts in several regions, more than two); each substring is processed (for instance, rewritten) by the rules in the region where it lies. There are at least two technical questions here: how to define the moving of strings through membranes? how to define the result of a computation?

p. A fruitful idea seems to be that of combining symbol-object and string-object P systems. In symbol-object systems we process multisets of symbols and the result of a computation is a number or a vector of natural numbers. In string-object systems we process strings in a way quite similar to language theory (to grammar system theory, because of distribution), without using multisets, and the result of a computation is a language. Let us consider multisets of strings, processed by rewriting, splicing, or by other string operations, but always taking into account the number of copies of each string. (For instance, when a derivation step of the form $x = x_1Ax_2 \Longrightarrow x_1ux_2 = y$ is performed, the number of copies of the string x is decreased by one and the number of copies of the string y is increased by one.) Consider as the result of a computation the number of strings present at the end of the computation in a specified membrane.

Of course, because we need to change the number of strings from a step to another one, rewriting operations (or splicing operations which recombine two strings and produce two new strings) are not enough, we need further operations. Some possibilities are suggested by the biology of the cell (for instance, the DNA biochemistry): cut a string into two new strings, duplicate a string, merge two strings into one new string; we can decrease the number of strings also by sending them out of the system. The study of such systems was already started in [P3], but only a few preliminary results were found: computational completeness and the possibility of solving the Hamiltonian Path Problem in quadratic time and the SAT problem in linear time. The used operations were reduplication, splitting, mutation, and crossing-over at given places. These operations have correspondents in the DNA biochemistry and similar operations are also used in [21]; crossing-over means passing from x_1zx_2, y_1zy_2 to x_1zy_2, y_1zx_2, because of the block z (this corresponds to the operation in [20] and is a restricted case of the splicing).

A particular problem in this framework: consider only string-objects of a bounded length. Does the maximal length induce an infinite hierarchy of the number sets computed by systems of a given type?

q. Generalizing the passing from symbol-objects to string-objects, an immediate further step is to consider still more complex structures for describing the objects, such as trees, graphs of arbitrary forms, arrays, etc. Such generalizations are classic in language theory (graph grammar area, array grammars, and even picture grammars are well developed domains); also in the DNA Computing were tried such generalizations, for instance, considering the splicing operation for trees and arrays (references can be found in [17]). In relation with the previously mentioned research topic of exporting membrane computing basic ingredients to other domains, it is natural to start by considering usual systems with complex object descriptions. Maybe the complexity of objects (of data structures we use) can compensate for using systems of a simple form, closer to "reality" and "easier to implement".

r. Of a definite interest are the research topics related to "applications" of P systems. A theoretical application, done only *in info*, is also called application here. This is the case with solving SAT (in [P3] and [P18]), the Hamiltonian Path Problem (in [P3] and [P8]), and the Node Covering Problem (in [P8]) in linear time, or to break DES (in [P10]). Find further hard (NP-complete) problems which can be "solved" in the membrane computing framework in a polynomial time (in a direct manner, not by a polynomial reduction to the NP-complete problems mentioned above). What about the primality question, can it be decided in a linear time? P systems deal with numbers, so it is expected that number theory questions are good candidates of problems to be addressed in the P area.

s. When attacking the previous NP-complete problems, one uses P systems with the possibility of dividing membranes. When a membrane is divided, its contents is almost completely replicated in the membranes obtained by division; in one step, does not matter how many objects and lower level membranes exist, all of them are replicated (in the case of [P18] in two copies, but in [P8] the number of copies is not bounded). This is a very powerful operation, not only because it enhances the parallelism (an exponential number of membranes can be obtained in a linear number of steps), but also because of this one-step replication of arbitrarily many objects. On the other hand, the initial model, that without membrane division already posseses a great amount of parallelism (somehow, of the type known from Lindenmayer systems). Is this parallelism sufficient in order to solve complex problems in an efficient way? Finding a specific NP-complete problem which can be solved in polynomial time by a P system without membrane division (and using symbol-objects) would be a very important result (the previous formulation suggests that I am a little bit pessimistic about this possibility: the symbols cannot contain "too much" information, as it is the case with membranes and with strings).

t. A "local" problem, related to the previous two points. In [P18] one deals with the case when the division of a membrane leads to two new membranes, while in [P8] one imposes no bound on the number of the new membranes. Is this latter feature necessary, or any system with, say, k-division can be simulated by a system with 2-division, maybe with a controlled slowdown? The intuition says that this would be the case: k descendant membranes of a given membrane can be obtained by $k-1$ divisions into two membranes; all other membranes and all objects not involved in these divisions can be "kept busy" for a number of steps depending on k by "timing rules" of the form $a \rightarrow a_1, a_1 \rightarrow a_2, \ldots, a_i \rightarrow a_{i+1}, \ldots, a_r \rightarrow a$. It happens that this intuition is not at all easy to be implemented. One of the main difficulties appears with the division of non-elementary membranes, when no object controls the division and, moreover, we do not know in advance how many copies of a membrane should be produced.

The 2-division systems can be considered as a normal form. A more general question for systems with membrane division is to look also for normal forms concerning other features, starting with the shape of the membrane structure; restricting the value of parameters related to the tree describing a membrane structure might be one of the directions for looking for such normal forms.

u. In what concerns the handling of membranes and the possibility of increasing their number, of interest can be one more idea: to create new membranes directly under the influence of objects, not starting from a previous membrane. In nature, it happens that membranes emerge in certain circumstances just by the organization of certain chemical compounds (lipid molecules). What about rules of the form $u \rightarrow [_i\ v\ ]_i$, which mean that the objects identified by u are transformed into the objects identified by v, surrounded by a fresh membrane having the label i. Because of the label, we precisely know the rules which can act in the region of this membrane i (we associate in advance a set of evolution rules with each membrane label).

This can be a very powerful computing tool: by a rule of the form $a \rightarrow aa$, in n steps we get 2^n copies of the object a; using then the rule $a \rightarrow [_i b]_i$ we can get 2^n copies of the membrane with the label i, which is very much similar to the way of producing exponentially many membranes in the case of systems with membrane division.

v. Also with respect to membrane handling, one can consider the possibility of communicating from a region to another one not only separate objects, but also membranes as a whole. For instance, by applying a rule of the form $[_i[_j\ ]_j]_i \rightarrow [_i\ ]_i[_j\ ]_j$ the membrane with the label j is sent out of the membrane with the label i (if the latter one is the skin membrane, then membrane j leaves the system). Of course, by moving membrane j outside membrane i, the contents of membrane j is moved as a whole (an arbitrarily large number of objects can be communicated at the same time outside membrane i; this is not at all similar to dissolving membrane i, because the communicated objects are now together in membrane j, which is placed in the membrane directly above membrane i, and both membranes i and j are still present in the system.

By using the previous rule in the reverse order (remember the reversibility problem; here we just look for rules which define the introduction of a membrane into another one), we can introduce membrane j, together with its contents, inside the region of membrane i. The computing possibilities open by using such rules, able to modify the very topology of the membrane structure, of "rewriting" it, seem to be very powerful (and intricate from a mathematical point of view).

w. Although this was mentioned several times before, I formulate as an explicit research topic the need of implementing P systems on the electronic computer, to simulate them by programs written in appropriate programming languages. Simulating parallelism and nondeterminism on a sequential deterministic machine can lose all the power and attractiveness of computing with P systems, while solving NP-complete problems in linear time means using an exponentially large space, but still the attempt should be done. The papers [P11] and [P29] have already tried this, but for systems without membrane division; moreover, the papers do not include actual programs (maybe the authors have such working programs and they have not distributed them).

Even if in this way we do not get an implementation of P systems *in silico*, we can get tools for related problems, such as simulating the life of a system, in the sense of Artificial Life. (For systems with only one membrane and with the number of possible objects bounded, this was already done – see [P30] and its references.)

x. Now, I come to an important point, the trade-off between *programmability, efficiency*, and *adaptability/learnability*, as pointed out by M. Conrad in several papers, see, e.g., [7], [8]. Looking for computing devices equal in power to Turing machines and also having universality properties (in the sense of the existence of universal devices in the considered class, which makes possible the programmability) is natural and seems desirable from a "classic" (= mathematical) point of view, but bio-computing seems to make possible an old dream of computer science, adaptable computers, which can "learn" both at the level of the hardware and at the level of the software. Even if the price of adaptability is the loss of programmability, maybe also of efficiency, it is highly possible that classes of problems exist for which it is preferable to have computers with an evolving hardware, adapted to the problem they have to solve. What this could mean in the membrane computing area is not at all clear, but the biological origin of P systems supports the question of investigating this topic. Suggestions from areas of Natural Computing where evolution and adaptation/learning are already central issues, like Neural Networks and Evolutionary Computing, will probably be useful.

y. In general, a more systematic osmosis with other areas of Natural Computing or with other approaches to distributed computing (such as systolic systems, actor systems, etc) is of interest and, expectedly, fruitful for both sides: ideas from other domains can be added (reformulated) to membrane computing area, while ideas from membrane computing area can (hopefully) be useful to other fields. P system theory has already imported many ingredients from grammar system theory (see, e.g., [9]) and from the DNA Computing area (see, e.g., [17]). A close correspondence with Cardelli's ambient calculus (see, [5], [6]) was found in [P28].

z. Membrane Computing comes from biology and in biology and biochemistry the processes are nondeterministic, the result is only approximately/probabilistically true. Up to now, only "crisp" mathematics was used in P system area (the same is in a great extent true also for DNA Computing). What about "approximate" mathematical approaches, using probabilities, fuzzy sets, or rough sets theory? (For the latter one, see [15]; the basic idea is to approximate a set from the interior and from the exterior, by converging unions of equivalence – or, more general, tolerance – classes.) What about "approximate" computing, whatever this can mean? All these can be reformulated in terms of *soft computing*, to which, in my opinion, Molecular Computing (DNA and membranes included) belongs.

There are no further letters in the alphabet, so I stop here...

Acknowledgments. These notes emerged mainly as a result of discussions with a series of people involved in the study of P systems. I only mention here, with indebtedness, a few of them (alphabetically): J. Castellanos (Madrid), E. Csuhaj-Varju (Budapest), R. Freund (Vienna), C. Martin-Vide (Tarragona), G. Mauri (Milano), V. Mitrana (Bucharest), A. Rodriguez-Paton (Madrid), G. Rozenberg (Leiden), Y. Sakakibara (Tokyo), A. Salomaa (Turku), Y. Suzuki (Tokyo), T. Yokomori (Tokyo).

Bibliography of P systems (May 2000)

[P1] A. Atanasiu, About a class of P systems, unfinished manuscript, 1999.

[P2] C. Calude, Gh. Păun, *Computing with Cells and Atoms*, Taylor and Francis, London, 2000 (Chapter 3: "Computing with Membranes").

[P3] J. Castellanos, Gh. Păun, A. Rodriguez-Paton, Computing with membranes: P systems with worm-objects, *IEEE Conf. SPIRE*, 2000.

[P4] J. Dassow, Gh. Păun, On the power of membrane computing, *J. of Universal Computer Sci.*, 5, 2 (1999), 33–49 (www.iicm.edu/jucs).

[P5] J. Dassow, Gh. Păun, Concentration controlled P systems, submitted, 1999.

[P6] R. Freund, Generalized P systems, *Fundamentals of Computation Theory, FCT'99*, Iaşi, 1999, (G. Ciobanu, Gh. Păun, eds.), *LNCS* 1684, Springer, 1999, 281–292.

[P7] R. Freund, Generalized P systems with splicing and cutting/recombination, *Workshop on Formal Languages*, FCT99, Iaşi, 1999.

[P8] S. N. Krishna, R. Rama, A variant of P systems with active membranes: Solving NP-complete problems, *Romanian J. of Information Science and Technology*, 2, 4 (1999), 357–367.

[P9] S. N. Krishna, R. Rama, On the power of P systems with sequential and parallel rewriting, manuscript, 1999.

[P10] S. N. Krishna, R. Rama, Computing with P systems, submitted, 2000.

[P11] M. Maliţa, Membrane computing in Prolog, manuscript, 1999.

[P12] C. Martin-Vide, V. Mitrana, P systems with valuation, submitted, 2000.

[P13] C. Martin-Vide, Gh. Păun, Computing with membranes: One more collapsing hierarchy, *Bulletin of the EATCS*, to appear.

[P14] A. Păun, M. Păun, On the membrane computing based on splicing, submitted, 2000.

[P15] Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61 (2000), in press, and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).

[P16] Gh. Păun, Computing with membranes. An introduction, *Bulletin of the EATCS*, 67 (Febr. 1999), 139–152.

[P17] Gh. Păun, Computing with membranes – A variant: P systems with polarized membranes, *Intern. J. of Foundations of Computer Science*, 11, 1 (2000), 167–182.

[P18] Gh. Păun, P systems with active membranes: Attacking NP complete problems, *J. Automata, Languages and Combinatorics*, in press, and *Auckland University, CDMTCS Report* No 102, 1999 (www.cs.auckland.ac.nz/CDMTCS).

[P19] Gh. Păun, Computing with membranes. A correction, two problems, and some bibliographical remarks, *Bulletin of the EATCS*, 68 (1999), 141–144.

[P20] Gh. Păun, From cells to computers. Computing with membranes (P systems), submitted, 1999.

[P21] Gh. Păun, G. Rozenberg, A. Salomaa, Membrane computing with external output, *Fundamenta Informaticae*, 41, 3 (2000), 259–266.

[P22] Gh. Păun, Y. Sakakibara, T. Yokomori, P systems on graphs of restricted forms, submitted, 1999.

[P23] Gh. Păun, G. Thierrin, Multiset processing by means of systems of finite state transducers, *Pre-Proc. of Workshop on Implementing Automata* WIA99, Potsdam, August 1999, Preprint 5/1999 of Univ. Potsdam (O. Boldt, H. Jürgensen, L. Robbins, eds.) XV 1-17, and *Auckland University, CDMTCS Report* No 101, 1999 (www.cs.auckland.ac.nz/CDMTCS).

[P24] Gh. Păun, T. Yokomori, Membrane computing based on splicing, *Preliminary Proc. of Fifth Intern. Meeting on DNA Based Computers* (E. Winfree, D. Gifford, eds.), MIT, June 1999, 213–227.

[P25] Gh. Păun, T. Yokomori, Simulating H systems by P systems, *Journal of Universal Computer Science*, 6, 2 (2000), 178–193 (www.iicm.edu/jucs).

[P26] Gh. Păun, S. Yu, On synchronization in P systems, *Fundamenta Informaticae*, 38, 4 (1999), 397–410.

[P27] I. Petre, A normal form for P systems, *Bulletin of the EATCS*, 67 (Febr. 1999), 165–172.

[P28] I. Petre, L. Petre, Mobile ambients and P systems, *Workshop on Formal Languages*, FCT99, Iaşi, 1999, *J. Universal Computer Sci.*, 5, 9 (1999), 588–598.

[P29] Y. Suzuki, H. Tanaka, On a LISP implementation of a class of P systems, *Romanian J. of Information Science and Technology*, 3, 2 (2000).

[P30] Y. Suzuki, J. Takabayashi, H. Tanaka, Investigation of an ecological system by using an abstract rewriting system on multisets, in *Recent Topics in Mathematical and Computational Linguistics* (Gh. Păun, ed.), The Publ. House of the Romanian Academy, Bucharest, 2000, 300–309.

[P31] Cl. Zandron, Cl. Ferretti, G. Mauri, Two normal forms for rewriting P systems, submitted, 2000.

[P32] Cl. Zandron, Cl. Ferretti, G. Mauri, Priorities and variable thickness of membranes in rewriting P systems, submitted, 2000.

References

[1] J. P. Banâtre, A. Coutant, D. Le Metayer, A parallel machine for multiset transformation and its programming style, *Future Generation Computer Systems*, 4 (1988), 133–144.

[2] C. Bennett, Logical reversibility of computation, *IBM J. Res. Dev.*, 17 (1973), 525–532.

[3] G. Berry, G. Boudol, The chemical abstract machine, *Theoretical Computer Sci.*, 96 (1992), 217–248.

[4] C. S. Calude, Gh. Păun, Global syntax and semantics for recursively enumerable languages, *Fundamenta Informaticae*, 4, 2 (1981), 245–254.

[5] L. Cardelli, Abstractions for mobile computation, in *Secure Internet Programming* (J. Vitek, Ch. Jensen, eds.), *Lecture Notes in Computer Science*, 1603, Springer-Verlag, 1999.

[6] L. Cardelli, A. Gordon, Mobile ambients, in *Proceedings of FoSSaCS'98* (M. Nivat, ed.), *Lecture Notes in Computer Science*, 1378, Springer-Verlag, 1998, 140–155.

[7] M. Conrad, Information processing in molecular systems, *Currents in Modern Biology*, 5 (1972), 1–14.

[8] M. Conrad, The price of programmability, *The Universal Turing Machine: A Half-Century Survey* (R. Herken, ed.), Kammerer and Unverzagt, Hamburg, 1988, 285–307.

[9] E. Csuhaj-Varju, J. Dassow, J. Kelemen, Gh. Păun, *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*, Gordon and Breach, London, 1994.

[10] J. Dassow, V. Mitrana, On some operations suggested by genome evolution, *Second Pacific Conf. on Biocomputing*, Hawaii, 1997.

[11] J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.

[12] J. Gruska, Descriptional complexity of context-free languages, *Proc. Math. Found. Computer Sci. Conf.*, High Tatras, 1973, 71–83.

[13] C. G. Langton, Artificial Life, in vol. *Artificial Life, II* (C. G. Langton, C. Taylor, J. D. Farmer, S. Rasmussen, eds.), Proc. of the Workshop on Artificial Life, Santa Fe, 1990, Santa Fe Institute Studies in the Science of Complexity, Proc. vol. X, Addison-Wesley, 1990, 1–47.

[14] S. S. Mader, *Biology* (Fifth Edition), McGraw-Hill, Boston, 1996 (Chapter: *Membrane structure and function*, 84–102).

[15] Z. Pawlak, *Rough Sets. Theoretical Aspects of Reasoning about Data*, Kluwer, Dordrecht, 1992.

[16] Gh. Păun, *Marcus Contextual Grammars*, Kluwer, Boston, Dordrecht, 1997.

[17] Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.

[18] Y. Rogozhin, Small universal Turing machines, *Theoretical Computer Sci.*, 168 (1996), 215–240

[19] G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, 3 volumes, Springer-Verlag, Berlin, 1997.

[20] M. P. Schützenberger: On finite monoids having only trivial subgroups. *Inform. Control*, 8 (1965), 190–194

[21] M. Sipper, Studying Artificial Life using a simple, general cellular model, *Artificial Life Journal*, 2, 1 (1995), 1–35.

[22] A. Syropoulos, A note on bags, basic pairs and the chemical abstract machine, submitted, 1999.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 218 - 235.

Computing with P Systems [1]

Raghavan RAMA
Department of Mathematics,
Indian Institute of Technology, Madras, Chennai-36
Tamilnadu, India
E-mail: `ramar@acer.iitm.ernet.in`

Abstract. P systems, introduced by Gh. Păun form a new class of biologically inspired distributed computing models. Several variants of P Systems were already shown to be computationally universal. In this paper, we establish that rewriting P Systems with priorities and two membranes is computationally universal, improving the existing result that $RE \subseteq RP_3(Pri)$. We give a new model in P Systems stressing the importance of parallelism and investigate its power. We also propose a class of P Systems capable of solving NP-complete problems like HPP and NCD in linear time. We also show that this class of P Systems can break the most widely used cryptosystem, DES in linear time.

Key Words: Membrane Computing, P Systems, NP-completeness, Hamiltonian Path Problem, Node Covering Problem, DES

1 Introduction

The P Systems are a class of distributed parallel computing devices of a biochemical inspiration. In this system, one considers a membrane structure consisting of several cell membranes which are hierarchially embedded in the main membrane, called the skin membrane. P Systems can be used as a support for a computing device based on any type of objects and any type of evolution rules associated with them [1 - 9]. The set of objects can evolve in many ways defined by string processing rules : Rewriting (Sequential and Parallel), point mutations and so on. In a P System based on rewriting, the objects considered are strings and a string passes through membranes as a unique entity, its symbols do not follow different itineraries, as it was possible for the objects in a multiset [3, 5]. In a P System based on parallel rewriting, a string passes through as a unique entity, and evolves parallely. That is, every symbol in the entity is rewritten whenever it is possible to do so. The rewriting rules are rules of L- systems and hence the entity evolves parallely.
We consider here systematically the four possibilities : P Systems with or without priorities, with or without cooperation. The four hierarchies on the number of membranes used in a system are compared to languages generated by devices in the Chomsky hierarchy and L-Systems. A characterization of recursively enumerable languages is obtained in the case of P Systems based on sequential rewriting. The proof uses the same technique as in [3], but this time, the number of membranes used is still smaller : two.

[1]this work was done with S.N.Krishna

We show that DES can be broken in constant time by using the variant of P Systems proposed in [5]. Also, we propose a variant of P systems with active membranes, where we allow an elementary membrane divide into finitely many membranes. The skin membrane is never divided. We prove that in this framework the Hamiltonian Path Problem and the Node Cover Decision Problem can be solved in linear time. This variant being more powerful and more general than the one considered in [5], its computational universality follows immediately.

2 P Systems Based On Sequential/Parallel Rewriting

We refer the reader to [3, 4] for basic notions, notations, and results about P Systems. Here we directly introduce the class of systems we investigate.

Definition 2.1 *A P System based on sequential rewriting is a language generating mechanism*
$\Pi = (V, T, \mu, w_1, w_2, \ldots, w_m, (R_1, \rho_1), (R_2, \rho_2), \ldots, (R_m, \rho_m))$ *where*

(1) $m \geq 1$*;*

(2) V *is an alphabet (the total alphabet of the system);*

(3) $T \subseteq V$ *(the terminal alphabet);*

(4) μ *is a membrane structure;*

(5) $w_1, \ldots, w_m$ *are strings over* V*, associated with regions* $1, \ldots, m$ *of* μ *describing the finite languages over* V

(6) $R_1, R_2, \ldots, R_m$ *are finite sets of developmental rules, associated with regions* $1, \ldots, m$ *of* μ *of the following forms:*

  (a) Context-free evolution rules of the form $X \longrightarrow v(tar)$ *where* $tar \in \{here, out, in_m\}$, $X \in V, v \in V^*$*.(m is the label of a membrane with the obvious meaning that the string produced using this rule will go into the membrane indicated by tar)*

  (b) Non Context-free rules of the form $u \longrightarrow v(tar)$ *of radius greater than one,* $u, v \in V^*$*.*

(7) $\rho_1, \rho_2, \ldots, \rho_m$ *are partial order relations on* $R_1, R_2, \ldots, R_m$*, specifying priority relations among the rules of* R_i*.*

In P Systems based on sequential rewriting, the objects we consider are strings. We apply atmost one rule in every step. The parallelism refers to the fact that we are processing simultaneously all available strings by all applicable rules. If several rules can be applied to a string, at several places each, we take only one rule and only one possibility to apply it and consider the obtained string as the next state of the object described by the string.
A P System based on parallel rewriting is similar to the system based on sequential

rewriting, but the parallelism is in two ways: We process simultaneously all strings in all membranes by all applicable rules: We rewrite every symbol of the string which can be rewritten. Thus several rules are applied to a string; each symbol is processed by one rule at a time. If a symbol occurs at several places of the string and if different rules can be applied to it, then at each position we nondeterministically choose a rule and apply it. In P Systems based on parallel rewriting all symbols which can be replaced in a step are replaced. Consider a string $a_1 a_2 \dots a_n$ and suppose there are rules $a_i \longrightarrow b_i(tari)$ where $tari$ represents some membrane. If all symbols are targeted to different membranes, one membrane is chosen nondeterministically and the string is moved there. Otherwise, if there are k targets occurring in the rules of a_i, $n_1, n_2, \dots, n_k$ times each, the string moves to target i provided $n_i = max\{n_1, n_2, \dots, n_k\}$ and assuming that there are no wrong rules [6]. That is, there are no rules R_i of the form $a \longrightarrow v$ with v introducing $b(in\ j)$ where j is not a membrane placed immediately inside i. Always, the string moves to a target only after all possible symbols are rewritten.

We denote by $ERP(\alpha, \beta)$ ($EPP(\alpha, \beta)$) the family of languages generated by P Systems based on sequential (parallel) rewriting with external output, $\alpha \in \{Pri, nPri\}$, $\beta \in \{Coo, nCoo\}$. If atmost m membranes have been used, then we write $ERP_m(\alpha, \beta)$ and $EPP_m(\alpha, \beta)$ for the family of languages generated by P Systems based on sequential and parallel rewriting respectively. The language generated by both the systems is the set of all those strings over terminals which go out of the system .

Theorem 2.1 $CF = ERP_1(nPri, nCoo)$

Proof : This result can be proved in exactly the same way as Theorem 2 in [3].

Theorem 2.2 $CF \subset ERP_1(Pri, nCoo)$

Theorem 2.3 $RE \subseteq ERP_1(nPri, Coo)$

Proof : The computational universality can be proved by considering a type-0 grammar G in Kuroda normal form.

Theorem 2.4 $RE \subseteq ERP_2(Pri, nCoo)$

Proof : Let G =(N, T, S, F)be a matrix grammar in binary normal form, $N = N_1 \cup N_2 \cup \{S, +\}, N_1 \cap N_2 = \phi$. Then the rules of F are of the following forms:

(1) $(S \longrightarrow XA), X \in N_1, A \in N_2$.

(2) $(X \longrightarrow Y, A \longrightarrow x), X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*$.

(3) $(X \longrightarrow Y, A \longrightarrow +), X, Y \in N_1, A \in N_2$, and $+$ is a special symbol.

(4) $(X \longrightarrow \lambda, A \longrightarrow x), X \in N_1, A \in N_2, x \in T^*$.

There is only one matrix of type 1, matrices of type 3 correspond to appearance checking rules. The symbol $+$ is a trap symbol, once introduced, it is never removed. All derivations can terminate only by a rule of type4. Assume that there are k matrices, numbered $m_1, m_2, \ldots, m_k$.
We construct the P System $\Pi = (V, T, [_1[_2]_2]_1, w_1, w_2, (R_1, \rho_1), (R_2, \rho_2))$ where $V = N_1 \cup N_2 \cup \{X_i, X_i' \mid 1 \leq i \leq k, X \in N_1\} \cup \{A_i, A_i' \mid 1 \leq i \leq k, A \in N_2\} \cup \{i, i', i'' \mid 1 \leq i \leq k\}$.
$w_1 = XA$ such that $(S \longrightarrow XA)$ is a matrix in F, $w_i = \phi, i \neq 1$. The rules are of the following forms:
R_1 :
$r_1 : \{X \longrightarrow Y_i \mid m_i : (X \longrightarrow Y, A \longrightarrow x)$ is of type $2\}$
$r_2 : \{X \longrightarrow Y_i' \mid m_i : (X \longrightarrow Y, A \longrightarrow +)$ is of type $3\}$
$r_3 : \{X \longrightarrow i' \mid m_i : (X \longrightarrow \lambda, A \longrightarrow x)$ is of type $4\}$
$r_4 : \{Y_i \longrightarrow i'Y_i \mid Y \in N_1, 1 \leq i \leq k\}$.
$r_5 : \{Y_i' \longrightarrow i''Y_i' \mid Y \in N_1, 1 \leq i \leq k\}$
$r_6 : \{i \longrightarrow \lambda \mid 1 \leq i \leq k\}$
$r_7 : \{A \longrightarrow A_i(in2) \mid m_i : (X \longrightarrow Y, A \longrightarrow x) or (X \longrightarrow \lambda, A \longrightarrow x)$ is a matrix of type 2 or 4 $\}$
$r_8 : \{A \longrightarrow + \mid m_i : (X \longrightarrow Y, A \longrightarrow +)$ is a matrix of type 3 $\}$
$r_9 : \{a \longrightarrow a(out) \mid a \in T\}$
$r_{10} : \{+ \longrightarrow +\}$
$r_{11} : \{X \longrightarrow + \mid X \in N_1$ and there does not exist any m_i containing a rule for $X\}$
$r_{12} : \{A \longrightarrow + \mid A \in N_2$ and there does not exist any m_i containing a rule for $A\}$

$$\rho_1 : \{r_6 > r_i, i = 1, \ldots, 5, 7, \ldots, 12, \quad r_1, r_2, r_3, r_4, r_{11} > r_7, \quad r_8 > r_5\}$$

R_2 :
$r_1 : \{A_i \longrightarrow iA_i' \mid A \in N_2\}$
$r_2 : \{i' \longrightarrow i \mid 1 \leq i \leq k\} \cup \{i'' \longrightarrow \lambda\}$ $r_2' : \{Y_i' \longrightarrow Y \mid 1 \leq i \leq k\}$
$r_3 : \{Y_i \longrightarrow Y(in1) \mid Y \in N_1\}$
$r_4 : \{+ \longrightarrow +\}$
$r_i : \{A_i' \longrightarrow x \mid m_i : (X \longrightarrow Y, A \longrightarrow x)$ is of type 2 $\} \cup$
$\{A_i' \longrightarrow x(in1) \mid m_i : (X \longrightarrow \lambda, A \longrightarrow x)$ is of type $4\} \cup$
$\{A_i' \longrightarrow + \mid m_i : (X \longrightarrow Y, A \longrightarrow +)$ is of type 3 $\}$, $1 \leq i \leq k$
$r_j' : \{j \longrightarrow + \mid 1 \leq j \leq k\}$

$\rho_2 : \{r_2, r_2' > r_1, r_3, r_4, r_i, r_j', \quad r_1, r_2, r_i, r_j', r_4 > r_3, \quad r_j' > r_i, i \neq j\}$
The system works as follows: Assume that in membrane 1, we have a string of the form $Xw, (X \in N_1, w \in (N_2 \cup T)^*)$. In membrane 1, one simulates matrix m_i as follows: First, the nonterminal from N_1 is rewritten along with the index i of the matrix, and then the nonterminal from N_2 is rewritten and the string goes to membrane 2, in case of a matrix of type 2 or 4. In case of an checking rule, the rule $A \longrightarrow +$ is applied and the computation never halts.
In membrane 2, we have to rewrite the rule corresponding to A_i, A_i', as this rule has higher priority than r_3 , which is the only rule to quit the membrane. A_i is first

rewritten as iA_i' and the rule corresponding to A_i' is applied if there does not exist two different indices i,j in the membrane. That is, A_i' is rewritten if matrix m_i has been correctly simulated. If there exists two different indices in the string, which means the simulation has not been correctly done, the rule $r_j' : j \longrightarrow +$ is applied and the computation never halts. If the simulation of the matrix is done correctly, the string goes to membrane 1 using r_3 or r_i according as a matrix of type 2 or 4 has been simulated. So, if the simulations are done correctly, we end up in membrane one with a terminal string which belongs to $L(G)$. This string can go out using $a \longrightarrow a(out), a \in T$. Hence, the language generated by the system is same as $L(G)$.

Remark 2.1 *We do not know whether this result is optimal. Whether a characterization of RE can be obtained with one membrane is left as an open problem.*

Theorem 2.5 $EOL = EPP_1(nPri, nCoo)$ *and* $E0L \subset EPP_1(Pri, nCoo)$

Theorem 2.6 $RE \subseteq EPP_2(Pri, nCoo)$

Proof : Let G be a matrix grammar in binary normal form. Assume that the matrices are numbered from 1 to k. We construct the parallel rewriting P System $\Pi = (V, T, [_1[_2]_2]_1, w_1, w_2, (R_1, \rho_1), (R_2, \rho_2))$,
$V = N_1 \cup N_2 \cup \{X_i X_i' \mid X \in N_1, 1 \leq i \leq k\} \cup \{A_i, A_i' \mid A \in N_2\} \cup \{A' \mid A \in N_2\} \cup \{i, i', i'' \mid 1 \leq i \leq k\}$
$w_1 = XA$ such that $(S \longrightarrow XA)$ is the initial matrix of G $w_2 = \phi$.
The rules are of the following forms:
R_1 :
$r_1 : \{X \longrightarrow i'Y_i(in2) \mid m_i : (X \longrightarrow Y, A \longrightarrow x)$ is of type 2 in $G\}$
$r_2 : \{X \longrightarrow i''Y_i'(in2) \mid m_i : (X \longrightarrow Y, A \longrightarrow +)$ is of type 3 in $G\}$
$r_3 : \{X \longrightarrow i'(in2) \mid m_i : (X \longrightarrow Y, A \longrightarrow x)$ is of type 4 in $G\}$
$r_4 : \{a \longrightarrow a(out) \mid a \in T\}$
$r_5 : \{X \longrightarrow + \mid X \in N_1$ and there does not exist any m_i containing a rule for $X\}$
$r_6 : \{A \longrightarrow + \mid A \in N_2$ and there does not exist any m_i containing a rule for $A\}$
$r_A : \{A \longrightarrow A_i(in2) \mid A \in N_2, m_i$ is a matrix of type 2, 3 or 4 in G and contains a rule for $A\}$
$r_A' : \{A_i \longrightarrow A_i(in2)\}$
$r_A'' : \{A \longrightarrow A(in2) \mid A \in N_2\}$
$r_B : \{B \longrightarrow B(in2) \mid B \in N_2, B \neq A\}$
$r_+ : \{+ \longrightarrow +\}$
$\rho_1 : \{r_A' > r_A\}$
R_2 :
$r_1 : \{i' \longrightarrow i \mid 1 \leq i \leq k\} \cup \{i'' \longrightarrow \lambda\} \cup \{Y_i' \longrightarrow Y \mid Y \in N_1\}$
$r_2 : \{Y_i \longrightarrow Y \mid Y \in N_1\}$
$r_3 : \{Y \longrightarrow Y(in1) \mid Y \in N_1\}$
$r_i : \{A_i' \longrightarrow x(in1) \mid m_i : (X \longrightarrow Y, A \longrightarrow x)$ or $(X \longrightarrow \lambda, A \longrightarrow x)$ is a matrix of type 2 or 4 in $G\}$ $\cup\{A_i' \longrightarrow +(in1) \mid m_i : (X \longrightarrow Y, A \longrightarrow +)$ is a matrix of type 3 in $G\} \cup \{i \longrightarrow \lambda \mid 1 \leq i \leq k\}$
$r_{A_i} : \{A_i \longrightarrow iA_i'\}$
$r_B : \{B \longrightarrow B' \mid B \in N_2\}$

$r'_B : \{B' \longrightarrow B(in1) \mid B \in N_2\}$
$r'_j : \{j \longrightarrow + \mid 1 \leq j \leq k\}$
$\rho_2 : \{r'_j > r_i, i \neq j\}$
The system works as follows : In the initial configuration, we have the string XA in membrane 1. Then, rules are applied to X and A and, as both targets are the same, viz membrane 2, the string is placed in membrane 2. There are two steps of rewriting taking place in membrane two : In the first step, we rewrite Y_i, i', A_i and the symbols from N_2. As all the targets are the same,the string remains here itself. In the next step, we rewrite i, A'_i, B' and Y giving priority to r'_j over r_i. This checks whether any rule has been wrongly applied, that is whether the chosen symbol $A \in N_2$ does not correspond to $X \in N_1$. If the simulation has been done correctly, the string is placed in membrane 1 since, a majority of the rules (except r'_j) are targeted to membrane 1. Assume that in membrane 1, we have a string of the form Yw, $Y \in N_1, w \in (N_2 \cup T)^*$. The rule corresponding to Y is applied, and also to the symbol $A \in N_2$ which corresponds to Y. All other symbols of N_2 are rewritten using r_B, and if the symbol A occurs more than once, it is rewritten using r''_A, since r'_A has higher priority than r_A and, since A can be rewritten in that step. After this step, the string is placed in membrane 2 and the computation proceeds in the manner described above. The rules $a \longrightarrow a(out), a \in T$ take the string out of the system, and it is listed in the language if it is purely over T^*. The priorities ensure that all simulations are done correctly and hence, we collect outside the skin membrane exactly the terminal strings generated by the grammar G and hence, $L(\Pi) = L(G)$.

Remark 2.2 *We do not know whether this result is optimal. Whether a characterization of RE can be obtained using one membrane is left as an open problem.*

Theorem 2.7 $RE \subseteq EPP_1(nPri, Coo)$

3 The DES circuit

In this section, we consider the "known plaintext" attack of cryptanalysis, [10], for breaking the data encryption standard. It means that the cryptanalyst knows some of the pairs consisting of plain-text and corresponding cipher-text and, on the basis of this information, he/she is supposed to find the key. First we give a brief overview of the DES circuit. The reader may refer [10] for further details on DES. DES encrypts a 64 bit plain-text into a 64 bit cipher-text using a 56 bit key. The DES circuit consists of 16 rounds or levels. The encryption is carried out as follows: First a key of 56 bits is chosen, and eight bits in positions $8, 16, \ldots, 64$ are added to the key, to check that each byte is of odd parity. The bits added are determined by the original 56 random bits, now in positions $1, 2, \ldots, 7, 9, \ldots, 15, \ldots, 57, \ldots, 63$ of the key. After subjecting these 56 bits to an initial permutation, we get two blocks C_0 and D_0 each of 28 bits each. Having constructed C_{n-1} and D_{n-1}, $n = 1, \ldots, 16$, C_nD_n is obtained from $C_{n-1}D_{n-1}$ by a rotation of bits which is predetermined. From C_nD_n, $n = 1, \ldots, 16$ we construct K_n, each of 48 bits by omitting certain bits from C_nD_n. Now consider the 64 bit plain text. Subjecting this to an *initial* *permutation*(*), we get two blocks L_0 and R_0, which are the lower and higher order bits of the permuted plain text. After computing L_{n-1} and R_{n-1}, $n = 1, \ldots, 16$ we obtain L_nR_n by applying the rules $L_n = R_{n-1}$ and $R_n = L_{n-1} \bigoplus f(R_{n-1}, K_n)$ where $\bigoplus$ denotes bit-by-bit addition modulo 2. Now we see how $f(R_{n-1}, K_n)$ is calculated. K_n is of 48 bits, R_{n-1} is of 32 bits. We convert R_{n-1} into a 48 bit according to a permutation which is predetermined. Then the 2 blocks R_{n-1} and K_n are added bit by bit modulo 2, which results in a 48 bit block. Let this block be denoted by $B_1B_2 \ldots B_8$, each B_i of 6 bits. We convert each B_i into a 4 bit block using lookup tables $T_1, T_2, \ldots, T_8$. Then a permutation is applied to this 32 bit block and this resultant is denoted by $f(R_{n-1}, K_n)$. After this step, performing a bit by bit addition modulo 2 on $f(R_{n-1}, K_n)$ and R_{n-2}, we get R_n. The encrypted text is obtained by applying the inverse of the *initial permutation*(*) to $R_{16}L_{16}$.

4 P Systems with Active Membranes and Plan of DES Attack

In this section we propose a variant of P Systems with active membranes, for basic notions we refer the reader to [5]. Let $d \geq 1$ be a natural number. A *P system with active membranes and d-bounded membrane division* (in short, we say a *P system with d-bounded division*) is a construct

$$\Pi = (V, T, H, \mu, w_1, \ldots, w_m, R),$$

where:

(1) $m \geq 1$;

(2) V is an alphabet (the *total alphabet* of the system);

(3) $T \subseteq V$ (the *terminal* alphabet);

(4) H is a finite set of *labels* for membranes;

(5) μ is a *membrane structure*, consisting of m membranes, labeled (not necessarily in a one-to-one manner) with elements of H; all membranes in μ are supposed to be neutral;

(6) $w_1, \ldots, w_m$ are strings over V, describing the *multisets of objects* placed in the m regions of μ;

(7) R is a finite set of *developmental rules*, of the following forms:

(a) $[_h a \to v]_h^\alpha$, for $h \in H,\ a \in V,\ v \in V^*,\ \alpha \in \{+, -, 0\}$ (object evolution rules),

(b) $a[_h]_h^{\alpha_1} \longrightarrow [_h b]_h^{\alpha_2}$, where $a, b \in V,\ h \in H,\ \alpha_1, \alpha_2 \in \{+, -, 0\}$ (an object is introduced in the membrane),

(c) $[_h a]_h^{\alpha_1} \to [_h\,]_h^{\alpha_2} b$, for $h \in H,\ \alpha_1, \alpha_2 \in \{+,-,0\},\ a, b \in V$ (an object is sent out),

(d) $[_h a]_h^{\alpha} \to b$, for $h \in H,\ \alpha \in \{+,-,0\},\ a, b \in V$ (dissolving rules),

(e) $[_h a]_h^{\alpha} \to [_h a_1]_h^{\alpha_1} [_h a_2]_h^{\alpha_2} \cdots [_h a_n]_h^{\alpha_n}$, for $\alpha, \alpha_i \in \{+,-,0\},\ a \in V,\ a_i \in V^*,\ i = 1, \ldots, n,\ h \in H$, and $n \leq d$(division rules for elementary membranes),

(f) $[_{h_0} [_{h_1}\ ]_{h_1}^{\alpha_1} \cdots [_{h_k}\ ]_{h_k}^{\alpha_k} [_{h_{k+1}}\ ]_{k_{k+1}}^{\alpha_{k+1}} \cdots [_{h_n}\ ]_{h_n}^{\alpha_n}\ ]_{h_0}]^{\alpha_0}$
$\to [_{h_0} [_{h_1}\ ]_{h_1}^{\beta_1}\ ]_{h_0}^{\beta_0} \cdots [_{h_0} [_{h_k}\ ]_{h_k}^{\beta_k}\ ]_{h_0}^{\beta_0} [_{h_0} [_{h_{k+1}}\ ]_{h_{k+1}}^{\beta_{k+1}}]_{h_0}^{\beta_0} \cdots [_{h_0} [_{h_n}\ ]_{h_n}^{\beta_n}\ ]_{h_0}^{\beta_0}$,
for $k \geq 1,\ n > k,\ h_i \in H,\ 0 \leq i \leq n,\ n \leq d$, and there is $i, 1 \leq i \leq n-1$, such that
$\alpha_i, \alpha_{i+1} = \{+,-\}$; moreover, $\beta_j \in \{+,-,0\},\ 1 \leq j \leq n$ (division of non-elementary membranes).

The rules of types (a) to (d) are applied following the same principles as in [5]: in a maximally parallel manner (any objects which can evolve, should evolve). If a membrane with label h is divided by a rule of type (e), which involves an object a, then all other objects in membrane h which do not evolve are introduced in each of the resulting membranes h. Similarly, when using rule of type (f), the membranes which are not specified in the rule, that is different from $[_{h_1}\ ]_{h_1}, \ldots, [_{h_n}\ ]_{h_n}$, are reproduced in each of the resulting membranes with the label h, unchanged if no rule is applied to them (the contents of these membranes are reproduced unchanged in these copies, providing that no rule is applied to their objects).
Note that in a rule of type (f) at least two membranes in its left hand side should have opposite polarization. When applying a rule of type (f) or (e) to a membrane, if there are objects in this membrane which evolve by a rule of type(a), changing the objects, then in the new copies of the membrane, we introduce the results of evolution. The rules are applied "from bottom up", in one step, but first the rules of the innermost region and then level by level until the region of the skin membrane. The rules associated with a membrane h are used for all copies of this membrane, irrespective whether or not the membrane is an initial one or it is obtained by division. At one step, a membrane h can be the subject of only one rule of types (b) – (f). The skin membrane can never divide. As any other membrane, the skin membrane can be "electrically charged". During a computation, objects can leave the skin membrane (by means of rules of type (c)). The terminal symbols which leave the skin membrane are collected in the order of their expelling from the system, so a string is associated to a complete (that is, halting) computation; when several terminal symbols leave the system at the same time, then any ordering of them is accepted. In this way, a language is associated with Π, denoted by $L(\Pi)$, consisting of all strings which are associated with all complete computations in Π. The symbols not in T which leave the skin membrane as well as all symbols from T which remain in the system at the end of a halting computation are not considered in the generated strings; if a computation goes for ever, then it provides no output, it does not contribute to the language $L(\Pi)$.

A P system with 1-bounded membrane division has no division at all, so rules of types (e), (f) are not used. In [5] one considered P systems with 2-bounded division. It is clear that our variant is more general than the one considered in [5], which is known to be computationally universal. Consequently, also P systems with d-bounded division are computationally universal for all $d \geq 2$. Division of a membrane into d membranes (d $\geq$ 2) is biologically motivated from cell fragmentation, where a cell can divide into two or more number of cells.

We are now ready to explain our attack on DES. Given a pair (plain-text, cipher-text), we need to check through all 2^{56} keys to find out which key maps the plain text to the cipher text. We construct a P System with active membranes and 2-bounded division which can output the secret key, given a (plain-text, cipher-text) pair. The time is estimated here as the number of steps our system works. This means, this is a parallel time where each unit is the time of a "biological" step in the system, the time of using any rule, supposing that all rules take the same time to be applied.

Theorem 4.1 *Given an arbitrary (plain-text, cipher-text) pair, it is possible to break DES in time linear in the size of the given crypto-text, using P Systems with active membranes and 2-bounded division.*

Proof : DES is a block cipher which operates on blocks of data. So, to encrypt a big file just think of the whole file as a collection of 64 bit blocks. DES encrypts the 64 bit blocks, one at a time (using the same DES secret key for each 64 bit block). In particular, if the file size is N bits then there will be $\lfloor N/64 \rfloor$ blocks, each block having exactly 64 bits. Of course if N is not an exact multiple of 64 then we pad the file with zeroes at the end in order to get an exact multiple of 64. First we construct a P System which can find out the secret key, given any arbitrary (plain-text, cipher-text) pair.
We construct the P System $\Pi = (V, T, H, \mu, w_0, w_1, \ldots, w_{18}, w_{0'}, w_{1'} \ldots, w_{64'}, R)$ where
$V = \{a_i, t_i, f_i, b_i, b'_i, 0_i, 1_i \mid 1 \leq i \leq 56\} \cup \{k_{i,j} \mid 1 \leq i \leq 16, 1 \leq j \leq 56\} \cup$
$\{m_{0,i} \mid 1 \leq i \leq 64\} \cup \{c_i \mid 0 \leq i \leq 111\} \cup \{f_{i,j}, d_{i,j} \mid 0 \leq i \leq 15, 1 \leq j \leq 48\} \cup$
$\{g_{i,j}, g'_{i,j} \mid 0 \leq i \leq 15, 1 \leq j \leq 32\} \cup \{e'_j, e''_j \mid 1 \leq j \leq 64\}$
$\cup \{e_{i,j} \mid 1 \leq i \leq 16, 1 \leq j \leq 56\}$; $T = \{t\}$; $\mu = [_{0'}[_{64'} \cdots [_{1'}[_{18} \cdots [_1]_1^0 \cdots]_{18}^0]_{1'}^0]_{64'}^0]_{0'}^0$,
$H = \{0, 1, \ldots, 18, 0', 1', \ldots, 64'\}$
$w_0 = c_0 a_1 a_2 \ldots a_{56} m_{0,1} m_{0,2} \ldots m_{0,64}$, $w_{18} = e''_1 \ldots e''_{64}, w_i = \phi, i \neq 0, 18.$
Suppose we are given a (plain-text, cipher-text) pair denoted by $(m_{0,1} m_{0,2} \ldots m_{0,64}, e''_1 \ldots e''_{64})$.
The rules are as follows:
$[_0 c_i \longrightarrow c_{i+1}]_0^\alpha, 0 \leq i \leq 110, \alpha \in \{+, -, 0\}$
$[_0 c_{111}]_0^\alpha \longrightarrow t, \alpha \in \{+, -, 0\}$
$[_0 a_i]_0^0 \longrightarrow [_0 t_i]_0^+ [_0 f_i]_0^-, 1 \leq i \leq 56$
$[_{i+1}[_i]_i^+ [_i]_i^-]_{i+1}^0 \longrightarrow [_{i+1}[_i]_i^0]_{i+1}^+ [_{i+1}[_i]_i^0]_{i+1}^-$, for $0 \leq i \leq 18$,$1' \leq i \leq 62'$
$[_{64'}[_{63'}]_{63'}^+ [_{63'}]_{63'}^-]_{64'}^0 \longrightarrow [_{64'}[_{63'}]_{63'}^0]_{64'}^0 [_{64'}[_{63'}]_{63'}^0]_{64'}^0$
(we generate all the 2^{56} permutations of the key. t_i and f_i stand for 0 and 1 respectively. When the counter reaches 111, we have generated all the permutations, and hence, membrane 0 dissolves. The last rule indicates that the skin membrane cannot be divided.)
$[_1 t_i]_1^\alpha \longrightarrow [_0 b_i]_1^\alpha, 1 \leq i \leq 56, \alpha \in \{+, -, 0\}$

$[_1 f_i]_1^\alpha \longrightarrow [_0 b_i]_1^\alpha, 1 \le i \le 56, \alpha \in \{+,-,0\}$
(i.e, we are renaming t_i and f_i, so that we have a string of b_i, the bits being the same.)
$[_1 m_{0,j}]_1^0 \longrightarrow [_1 l_{0,k}]_1, 1 \le k \le 32, 1 \le j \le 64$.
$[_1 m_{0,j}]_1^0 \longrightarrow [_1 r_{0,k}]_1^0, 1 \le k \le 32, 1 \le j \le 64.$
(From the given plain text, we form two blocks L_0 and R_0, of 32 bits each, by a permutation which is predetermined. Here also, the bits themselves do not change, the only change made is in the ordering, by which we get two blocks L_0 and R_0)
$[_1 b_i]_1^0 \longrightarrow [_1 b'_i k_{1,j_1} k_{2,j_2} \dots k_{16,j_{16}}]_1^0, 1 \le i, j_1, \dots, j_{16} \le 56$
$[_j b'_i]_j^0 \longrightarrow [_j]_j^0 b'_i, 1 \le j \le 18, 1 \le i \le 56$
(From all the permutations of the key, we form C_1D_1, $\dots C_{16}D_{16}$. If the bit at position i in the key is placed at positions $j_1, j_2, \dots, j_{16}$ in $C_1D_1, C_2D_2, \dots, C_{16}D_{16}$, we have the above rule. So, after this step, $k_{i,1} \dots k_{i,56}$ represents C_iD_i, $1 \le i \le 16$. Also, $b'_i = b_i$, we are just renaming the key, we need b'_i as the output after a complete computation)
$[_1 k_{n,j}]_1^0 \longrightarrow [_1]_1^0, n > 1, 1 \le j \le 56$
$[_1 k_{1,j}]_1^0 \longrightarrow [_1]_1^0, 1 \le j \le 56, j = 9, 18, 22, 25, 35, 38, 43, 54$
$[_1 k_{1,j}]_1 \longrightarrow [_1]_1 e_{1,l}, 1 \le l \le 48, 1 \le j \le 56, j \ne 9, 18, 22, 25, 35, 38, 43, 54$
(For generating R_1, we need only K_1. So, we let $C_2D_2, \dots, C_{16}D_{16}$ go out of membrane 1. Now, the bits at positions 9, 18, 22, 25, 35, 38, 43, 54 of C_1D_1 are removed and the remaining bits permuted to get $e_{1,1} \dots e_{1,48}$ which represents K_1).
$[_1 l_{0,j}]_1^0 \longrightarrow [_1]_1^0 l_{0,j}, j = 1, \dots, 32.$
(L_0 goes out of membrane 1)
For $1 \le j \le 16$, we have the rules given below:
$[_j r_{j-1,1}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,1} d_{j-1,1} d_{j-1,2}, \quad d_{j-1,1} = r_{j-1,32}, d_{j-1,2} = r_{j-1,1}$
$[_j r_{j-1,2}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,2} d_{j-1,3}, \quad d_{j-1,3} = r_{j-1,2}$
$[_j r_{j-1,3}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,3} d_{j-1,4}, \quad d_{j-1,4} = r_{j-1,3}$
$[_j r_{j-1,4}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,4} d_{j-1,5} d_{j-1,6}, \quad d_{j-1,5} = r_{j-1,4}, d_{j-1,6} = r_{j-1,5}$
$[_j r_{j-1,5}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,5} d_{j-1,7} d_{j-1,8}, \quad d_{j-1,7} = r_{j-1,4}, d_{j-1,8} = r_{j-1,5}$
$[_j r_{j-1,6}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,6} d_{j-1,9}, \quad d_{j-1,9} = r_{j-1,6}$
$[_j r_{j-1,7}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,7} d_{j-1,10}, \quad d_{j-1,10} = r_{j-1,7}$
$[_j r_{j-1,8}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,8} d_{j-1,11} d_{j-1,12}, \quad d_{j-1,11} = r_{j-1,8}, d_{j-1,12} = r_{j-1,9}$
$[_j r_{j-1,9}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,9} d_{j-1,13} d_{j-1,14}, \quad d_{j-1,13} = r_{j-1,8}, d_{j-1,14} = r_{j-1,9}$
$[_j r_{j-1,10}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,10} d_{j-1,15}, \quad d_{j-1,15} = r_{j-1,10}$
$[_j r_{j-1,11}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,11} d_{j-1,16}, \quad d_{j-1,16} = r_{j-1,11}$
$[_j r_{j-1,12}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,12} d_{j-1,17} d_{j-1,18}, \quad d_{j-1,17} = r_{j-1,12}, d_{j-1,18} = r_{j-1,13}$
$[_j r_{j-1,13}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,13} d_{j-1,19} d_{j-1,20}, \quad d_{j-1,19} = r_{j-1,12}, d_{j-1,20} = r_{j-1,13}$
$[_j r_{j-1,14}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,14} d_{j-1,21}, \quad d_{j-1,21} = r_{j-1,14}$
$[_j r_{j-1,15}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,15} d_{j-1,22}, \quad d_{j-1,22} = r_{j-1,15}$
$[_j r_{j-1,16}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,16} d_{j-1,23} d_{j-1,24}, \quad d_{j-1,23} = r_{j-1,16}, d_{j-1,24} = r_{j-1,17}$
$[_j r_{j-1,17}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,17} d_{j-1,25} d_{j-1,26}, \quad d_{j-1,25} = r_{j-1,16}, d_{j-1,26} = r_{j-1,17}$
$[_j r_{j-1,18}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,18} d_{j-1,27}, \quad d_{j-1,27} = r_{j-1,18}$
$[_j r_{j-1,19}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,19} d_{j-1,28}, \quad d_{j-1,28} = r_{j-1,19}$
$[_j r_{j-1,20}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,20} d_{j-1,29} d_{j-1,30}, \quad d_{j-1,29} = r_{j-1,20}, d_{j-1,30} = r_{j-1,21}$
$[_j r_{j-1,21}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,21} d_{j-1,31} d_{j-1,32}, \quad d_{j-1,31} = r_{j-1,20}, d_{j-1,32} = r_{j-1,21}$

$[_j r_{j-1,22}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,22} d_{j-1,33},\;\; d_{j-1,33} = r_{j-1,22}$
$[_j r_{j-1,23}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,23} d_{j-1,34},\;\; d_{j-1,34} = r_{j-1,23}$
$[_j r_{j-1,24}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,24} d_{j-1,35} d_{j-1,36},\;\; d_{j-1,35} = r_{j-1,24}, d_{j-1,36} = r_{j-1,25}$
$[_j r_{j-1,25}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,25} d_{j-1,37} d_{j-1,38},\;\; d_{j-1,37} = r_{j-1,24}, d_{j-1,38} = r_{j-1,25}$
$[_j r_{j-1,26}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,26} d_{j-1,39},\;\; d_{j-1,39} = r_{j-1,26}$
$[_j r_{j-1,27}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,27} d_{j-1,40},\;\; d_{j-1,40} = r_{j-1,27}$
$[_j r_{j-1,28}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,28} d_{j-1,41} d_{j-1,42},\;\; d_{j-1,41} = r_{j-1,28}, d_{j-1,42} = r_{j-1,29}$
$[_j r_{j-1,29}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,29} d_{j-1,43} d_{j-1,44},\;\; d_{j-1,43} = r_{j-1,28}, d_{j-1,44} = r_{j-1,29}$
$[_j r_{j-1,30}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,30} d_{j-1,45},\;\; d_{j-1,45} = r_{j-1,30}$
$[_j r_{j-1,31}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,31} d_{j-1,46},\;\; d_{j-1,46} = r_{j-1,31}$
$[_j r_{j-1,32}]_j^0 \longrightarrow [_j]_j^0 r_{j-1,32} d_{j-1,47} d_{j-1,48},\;\; d_{j-1,47} = r_{j-1,32}, d_{j-1,48} = r_{j-1,1}$
(We convert R_{n-1} which is of 32 bits into a 48 bit block, so that we can calculate $f(R_{n-1}, K_n)$, where K_n is of 48 bits. Now, $d_{j,1} \dots d_{j,48}$ is of 48 bits, and it is passed out along with R_{j-1} to membrane $(j+1)$, where we compute R_j.)
The remaining rules are as follows:
$[_j k_{m,n}]_j^0 \longrightarrow [_j]_j^0 k_{m,n}, m > j-1, 2 \le j \le 16, 1 \le n \le 56, 1 \le m \le 16$
$[_j k_{j-1,n}]_j^0 \longrightarrow [_j]_j^0$, n= 9, 18, 22, 25, 35, 38, 43, 54.
$[_j k_{j-1,n}]_j^0 \longrightarrow [_j e_{j-1,l}]_j^0, 3 \le j \le 17, 1 \le n, \le 56, 1 \le l \le 48, n \ne 9, 18, 22, 25, 35, 38, 43, 54.$
(In membrane j, we need $C_{j-1}D_{j-1}$ and no other C_iD_i to compute R_{j-1}. So we let $k_{l,n}, n = 1, \dots, 56$, and $l > j-1$ to go out of membrane j. Then we convert $C_{j-1}D_{j-1}$ into K_{j-1})
$[_j d_{j-2,k}]_j^0 \longrightarrow [_j d_{j-2,k} \bigoplus e_{j-1,k} = f_{j-2,k}]_j^0$, for $2 \le j \le 17, 1 \le k \le 48$.
(For computing R_{j-1} in membrane j, we need to compute $f(R_{j-2}, K_j)$ first. That is why we compute f_j from the 48-bit block d_j)
For $2 \le j \le 17$,
$[_j f_{j-2,1}]_j^0 \longrightarrow [_j g'_{j-2,1} g'_{j-2,2} g'_{j-2,3} g'_{j-2,4}]_j^0$,
$[_j f_{j-2,2}]_j^0 \dots [_j f_{j-2,6}]_j^0 \longrightarrow [_j]_j^0$ where $g'_{j-2,1} g'_{j-2,2} g'_{j-2,3} g'_{j-2,4}$ is the four bit corresponding to the six bit $f_{j-2,1} \dots f_{j-2,6}$ from the lookup table T_1.
$[_j f_{j-2,7}]_j^0 \longrightarrow [_j g'_{j-2,5} g'_{j-2,6} g'_{j-2,7} g'_{j-2,8}]_j^0$,
$[_j f_{j-2,8}]_j^0 \dots [_j f_{j-2,12}]_j^0 \longrightarrow [_j]_j^0$
(Corresponding to lookup table T_2)
$[_j f_{j-2,13}]_j^0 \longrightarrow [_j g'_{j-2,9} g'_{j-2,10} g'_{j-2,11} g'_{j-2,12}]_j^0$,
$[_j f_{j-2,14}]_j^0 \dots [_j f_{j-2,18}]_j^0 \longrightarrow [_j]_j^0$
(Corresponding to lookup table T_3)
$[_j f_{j-2,19}]_j^0 \longrightarrow [_j g'_{j-2,13} g'_{j-2,14} g'_{j-2,15} g'_{j-2,16}]_j^0$,
$[_j f_{j-2,20}]_j^0 \dots [_j f_{j-2,24}]_j^0 \longrightarrow [_j]_j^0$
(Corresponding to lookup table T_4)
$[_j f_{j-2,25}]_j^0 \longrightarrow [_j g'_{j-2,17} g'_{j-2,18} g'_{j-2,19} g'_{j-2,20}]_j^0$,
$[_j f_{j-2,26}]_j^0 \dots [_j f_{j-2,30}]_j^0 \longrightarrow [_j]_j^0$
(Corresponding to lookup table T_5)
$[_j f_{j-2,31}]_j^0 \longrightarrow [_j g'_{j-2,21} g'_{j-2,22} g'_{j-2,23} g'_{j-2,24}]_j^0$,
$[_j f_{j-2,32}]_j^0 \dots [_j f_{j-2,36}]_j^0 \longrightarrow [_j]_j^0$
(Corresponding to lookup table T_6)
$[_j f_{j-2,37}]_j^0 \longrightarrow [_j g'_{j-2,25} g'_{j-2,26} g'_{j-2,27} g'_{j-2,28}]_j^0$,
$[_j f_{j-2,38}]_j^0 \dots [_j f_{j-2,42}]_j^0 \longrightarrow [_j]_j^0$

(Corresponding to lookup table T_7)
$[_j f_{j-2,43}]_j^0 \longrightarrow [_j g'_{j-2,29} g'_{j-2,30} g'_{j-2,31} g'_{j-2,32}]_j^0$,
$[_j f_{j-2,44}]_j^0 \cdots [_j f_{j-2,48}]_j^0 \longrightarrow [_j]_j^0$
(Corresponding to lookup table T_8)

For $2 \leq j \leq 17, 1 \leq k, l \leq 32$,
$[_j g'_{j-2,k}]_j^0 \longrightarrow [_j g_{j-2,l}]_j^0$
$[_j r_{j-3,k}]_j^0 \longrightarrow [_j r_{j-3,k} \bigoplus g_{j-2,k} = r_{j-1,k}]_j^0, 3 \leq j \leq 17, 1 \leq k \leq 32.$
$[_j r_{j-2,k}]_j^0 \longrightarrow [_j]_j^0 r_{j-2,k}, 2 \leq j \leq 16, 1 \leq k \leq 32$
(In membrane j, we are sending out R_{j-2} as it is required in membrane $(j+1)$ for producing $R_j = R_{j-2} \bigoplus f(R_{j-1}, K_j)$. We use R_{j-3} for producing R_{j-1} and the obtained R_{j-1} is converted to 48 bits and sent out to membrane $(j+1)$ for the production of R_j)
$[_2 l_{0,j}]_2^0 \longrightarrow [_2 r_{1,j}]_2^0$, where $r_{1,j} = l_{0,j} \bigoplus g_{0,j}, j = 1, \ldots, 32.$
Now in membrane 17, we have, in addition to the above rules,the following:
$[_{17} r_{16,j}]_{17}^0 \longrightarrow [_{17}]_{17}^0 e'_j, j = 1, \ldots, 32.$
$[_{17} r_{15,j}]_{17}^0 \longrightarrow [_{17}]_{17}^0 e'_{j+32}, j = 1, \ldots, 32.$
(Now $e'_1 \ldots e'_{64}$ is nothing but $R_{16} L_{16}$, since here also the bits are the same, the order is only changed. Note here that $r_{16,j} = e'_j$, and $r_{15,j} = e'_{j+32}$).
$[_{18} e'_j]_{18}^0 \longrightarrow [_{18} h_k]_{18}^0, 1 \leq j, k \leq 64$
(h_k is obtained from e'_j by applying the inverse of the initial permutation. Here also, only the order of the bits are changed, the bits are not changed.)
$[_{18} e''_j]_{18}^0 \longrightarrow [_{18}]_{18}^0 \mid e''_j - h_j \mid_j, 1 \leq j \leq 64.$
(So we have 0_j or 1_j, $1 \leq j \leq 64$. If we have zeroes alone, it means $h_1 h_2 \ldots h_{64}$ is the same as $e''_1 \ldots e''_{64}$, and this in turn means that b_i is the secret key.)
$[_{j'} 0_j]_{j'} \longrightarrow 0_j, 1 \leq j \leq 64$
(Membrane j' dissolves if the difference between the jth bits is zero.)
$[_{0'} b'_i]_{0'}^0 \longrightarrow [_{0'}]_{0'}^0, b'_i, 1 \leq i \leq 55,$
$[_{0'} b'_{56}]_{0'}^0 \longrightarrow [_{0'}]_{0'}^+, b'_{56}$
(If the given cipher text matches with the computed cipher text, then all membranes j', $1 \leq j \leq 64$ will dissolve. Then b'_i approaches the skin membrane and from there it can go out thereby, giving the information that $b_1 \ldots b_{56}$ is the secret key.)

Now we analyze the time complexity of the above procedure:
First, we generate all possible permutations of 56 bits, and this stops when the counter reaches 111. After this, membrane 0 dissolves and c_{111} is converted to t. In the next step, t_i and f_i are converted to b_i. So, the total time required is 113 steps. Next we consider the operations in membrane 1 and those going parallely with that.

(1) Conversion of the key into $C_1 D_1 \ldots C_{16} D_{16}$, and conversion of $m_{0,j}$ into $l_{0,k}$ and $r_{0,l}$ respectively. These two operations are carried out in parallel and take O(1) time.

(2) L_0, R_0, $C_2 D_2, \ldots C_{16} D_{16}$ and b'_i, $1 \leq i \leq 56$ go out of membrane 1 and $C_1 D_1$ is converted to K_1.

These operations are also done in time O(1).Hence the total time spent in membrane 1 along with the operations going parallely with that takes 2 steps.
Next, we consider the operations going parallely in membrane 2 and outside membrane 2.

(1) In membrane 2, R_0, $k_{n,j}$, b'_i go out, for $n \geq 2$, $1 \leq i \leq 56$ and $d_{0,j}$ is converted to $f_{0,j}$.

(2) In membrane 2, $f_{0,j}, 1 \leq j \leq 48$ is converted into $g'_{0,k}, 1 \leq k \leq 32$, according to the lookup tables, and in membrane 3, $k_{n,j}$, b'_i go out for $n > 2$, $1 \leq i \leq 56$. Also, C_2D_2 is converted to K_2.

(3) In membrane 2, $g'_{0,k}, 1 \leq k \leq 32$ is converted to $g_{0,k}, 1 \leq k \leq 32$, and in membrane 4, $k_{n,j}$, b'_i go out for n>3, $1 \leq i \leq 56$, and C_3D_3 is converted to K_3.

(4) R_1 is computed in membrane 2, and in membrane 5, $k_{n,j}$, b'_i go out for n>4, $1 \leq i \leq 56$, and C_4D_4 is converted to K_4.

(5) R_1 goes out of membrane 2 as R_1 and as a 48-bit block, and in membrane 6, $k_{n,j}$, b'_i go out for n>5,$1 \leq i \leq 56$ and C_5D_5 is converted to K_5.

Hence, the total time spent here is for 5 steps. Next, the operations in membrane 3 and the ones going parallely with it:

(1) In membrane 3, R_1 goes out and $d_{1,k}$ is converted to $f_{1,k}1 \leq k \leq 32$, and in membrane 7, $k_{n,j}$, b'_i go out for n>6, $1 \leq i \leq 56$. Also, C_6D_6 is converted to K_6.

(2) In membrane 3, $f_{1,k}$ is converted to $g'_{1,l}, 1 \leq k \leq 48, 1 \leq l \leq 32$, and in membrane 8, b'_i, $k_{n,j}$ go out for n>7, $1 \leq i \leq 56$ and C_7D_7 is converted to K_7.

(3) In membrane 3, $g'_{1,l}, 1 \leq l \leq 32$ is converted to $g_{1,j}, 1 \leq j \leq 32$, and in membrane 9, $k_{n,j}$ and b'_i go out for n>8, $1 \leq i \leq 56$, C_8D_8 is converted to K_8.

(4) R_0 is converted to R_2 in membrane 3, and in membrane 10, $k_{n,j}$ and b'_i go out for n>9, C_9D_9 is converted to K_9.

(5) R_2 goes out of membrane 3 as R_2 and as a 48-bit block, and in membrane 11,$k_{n,j}$ and b'_i go out for n>10, $1 \leq i \leq 56$, $C_{10}D_{10}$ is converted to K_{10}.

Hence, the total time required is again, 5 steps. Similarly, the time required in membranes $4, \ldots, 16$ is also 5 steps.
In membrane 17, we have the following sequence of events:
$d_{15,k}, 1 \leq k \leq 48$ is converted to $f_{15,k}$,$f_{15,k}$ is converted to $g'_{15,l}, 1 \leq l \leq 32$, $g'_{15,l}$ is converted to $g_{15,j}, 1 \leq j \leq 32$, R_{16} is computed, and $e'_1 \ldots e'_{64}$ is computed from R_{15} and R_{16}. Hence, the time required in membrane 17 is also 5 steps. Likewise, we have in membrane 18, $e'_1 \ldots e'_{64}$ is converted to $h_1 \ldots h_{64}$, e''_j is converted to $\mid e''_j - h_j \mid_j, 1 \leq j \leq 64$. The time required here is 2 steps. Finally we require time

corresponding to 65 steps to dissolve the 64 membranes $1', \ldots, 64'$, and to output the secret key. Hence, the total time required to find the secret key is 113 + 2 + 16* 5 + 2 + 65 = 262 steps. Now given an encrypted text of size N bits, we do the following steps:

(1) Divide the N bit text into blocks of size 64. If N is not an exact multiple of 64, pad it with zeroes so that we have k blocks each of size 64.

(2) Decrypt each block using the secret key obtained by the above procedure. Since the whole encryption is made public, once we know the secret key, decrypting each block takes time O(1). Hence the total time required to decrypt the given text is O([N/64]).

Therefore, the total time for breaking DES is 262+[N/64], which is linear in N.

5 The Hamiltonian Path Problem and Node Cover Decision Problem for Undirected Graphs

In this section, we consider two intractable problems, both of which can be solved in linear time by P systems with active membranes and bounded division.

The first problem is the Hamiltonian Path Problem for undirected graphs, known to be NP-complete. Given an undirected graph $G = (U, E)$, where U is the set of nodes and E the set of edges, the problem is to determine whether or not there exists a Hamiltonian path in G, that is, to determine whether or not there exists a path that passes through all the nodes in U exactly once.

Theorem 5.1 *The Undirected Hamilton Path Problem can be solved by a P system with bounded membrane division in a time which is linear in the number of nodes of the given graph.*

Proof : Let $G = (U, E)$ be an undirected graph with n nodes, $n \geq 2$. We construct the P system (with n-bounded division)

$$\Pi = (V, T, H, \mu, w_0, w_1, \ldots, w_{n+1}, R),$$

with the following components:

$$\begin{aligned} V &= \{a_i \mid 1 \leq i \leq n\} \cup \{c_i \mid 0 \leq i \leq 2n-1\} \cup \{1, 2, \ldots, n\} \cup \{t, f\}, \\ T &= \{t\}, \\ H &= \{0, 1, \ldots, n+1\}, \\ \mu &= [_{n+1} \cdots [_1 [_0\ ]^0_0]^0_1 \cdots]^0_{n+1}, \\ w_0 &= c_0 f,\ w_i = \lambda,\ \text{for } 1 \leq i \leq n+1, \end{aligned}$$

while the set R contains the following rules:

1. $[_0 c_i \rightarrow c_{i+1}]^\alpha_0$, for $0 \leq i < 2n-1$, and $\alpha \in \{+, -, 0\}$
(We count up to $2n-1$, which is the time required for the paths we look for to grow to length n, beginning from every node.)

2. $[_0 c_{2n-1}]_0^\alpha \rightarrow t$, for $\alpha \in \{+,-,0\}$
(After $2n-1$ steps from the beginning of the computation, the membranes with label 0 are dissolved.)

3. $[_0 f]_0^0 \rightarrow [_0 a_1]_0^+ [_0 a_2]_0^- \cdots [_0 a_n]_0^+$,
where the charge of the last membrane is $+$ or $-$ depending on whether n is odd or even.
(Division rule for membrane 0 when it is electrically neutral. By this rule, we get access to all n nodes and hence the adjacencies corresponding to each node.)

4. $[_0 a_i]_0^0 \rightarrow [_0 i a_{j_1}]_0^+ [_0 i a_{j_2}]_0^- \cdots [_0 i a_{j_k}]_0^+$, if node i in G has the nodes $j_1, \ldots, j_k$ adjacent to it, $1 \leq j_l \leq n, 1 \leq l \leq k$; the charge of the last membrane is $+$ or $-$ depending on whether i has an odd or even number adjacent nodes, while $1 \leq i, k \leq n$. If i has only one node j adjacent to it, then the rule is
$[_0 a_i]_0^0 \rightarrow [_0 i a_j]_0^0$
(By using these rules, we generate all possible paths starting from every node.)

5. $[_1 a_i \rightarrow i]_1^\alpha$, for $1 \leq i \leq n,\ \alpha \in \{+,-,0\}$
(After dissolving membrane 0, hence after growing paths of length n, the object a_i reaches membrane 1 and its own index, i, is introduced in the path.)

6. $[_i i]_i^\alpha \rightarrow i$, for $1 \leq i \leq n,\ \alpha \in \{+,-,0\}$
(Membrane i is dissolved if node i is present in the path.)

7. $[_1 a_1]_1^\alpha \rightarrow 1$, for $\alpha \in \{+,-,0\}$.
(Membrane 1 is dissolved when replacing a_1 by 1.)

8. $[_{n+1} t]_{n+1}^0 \rightarrow [_{n+1}\ ]_{n+1}^+ t$
(The object t leaves the skin membrane if all lower membranes are dissolved, which means that the graph has a Hamiltonian path.)

9. $[_{i+1} [_i\ ]_i^+ [_i\ ]_i^- \cdots [_i\ ]_i^+]_{i+1}^0 \rightarrow [_{i+1} [_i\ ]_i^0]_{i+1}^+ [_{i+1} [_i\ ]_i^0]_{i+1}^- \cdots [_{i+1} [_i\ ]_i^0]_{i+1}^+$, for $0 \leq i \leq n-2$,
$[_n [_{n-1}\ ]_{n-1}^+ [_{n-1}\ ]_{n-1}^- \cdots [_{n-1}\ ]_{n-1}^+]_n^0 \rightarrow [_n [_{n-1}\ ]_{n-1}^0]_n^0 [_n [_{n-1}\ ]_{n-1}^0]_n^0 \cdots [_n [_{n-1}\ ]_{n-1}^0]_n^0$
(Division rules for membranes labeled with $0, 1, \ldots, n$; the opposite polarization introduced when dividing a membrane with label 0 is propagated from lower levels to upper levels of the membrane structure and the membranes are continuously divided until dividing also membrane n – which will get neutral charge).

Clearly, the system Π has a d-bounded division, for $d = n$.

From the above explanations, it is easy to see that

$$L(\Pi) = \begin{cases} \{t\}, & \text{if the given graph has a Hamiltonian path,} \\ \emptyset, & \text{otherwise.} \end{cases}$$

In $2n-1$ steps, we generate all possible paths of length n starting from every node. In the next step, membrane 0 is dissolved, then, at the next step, each object a_i is replaced by i; at the same time, rules of type 6 are used, dissolving the membranes

with label 1 if node 1 is present in the corresponding path. Note that in the case when membrane 1 containes the object a_1, the dissolvation is done by the rule $[_1 a_1]_1 \to 1$. In the next $n-1$ steps, membranes 2 to n are dissolved if nodes 2 to n are present in the generated path. If all membranes are dissolved by a given path, this means that it is a Hamiltonian path; in such a case – and only in such a case – the object t reaches the skin membrane. Then it takes one more step to output t. Consequently, in $3n+1$ steps, we get the answer to the question whether G has a Hamiltonian path or not by checking the emptiness of the language $L(\Pi)$ (more precisely, by observing whether or not the system produces a symbol t at the moment $3n+1$). Note that in the above construction, we are using all rules of types (a)–(f), except (b). The above construction also solves the Hamiltonian Path Problem for directed graphs.
We next consider the Node Cover Decision problem. Given a graph G with n nodes and m edges, the Node Covering Problem asks whether or not there exists a node cover for G of size k, where k is a given integer less than or equal to n. We show that also this problem can be solved in linear time using the P systems introduced above.

Theorem 5.2 *The Node Cover Problem can be solved by the P systems with bounded membrane division in a time which is linear in the number of edges and the number of covering nodes.*

Proof : Let $G = (U, E)$ be a graph with n nodes and m edges. We construct the P system

$$\Pi = (V, T, H, \mu, w_1, \ldots, w_{m+1}, R),$$

where:

$$\begin{aligned} V &= \{a_i \mid 1 \le i \le n\} \cup \{c_i \mid 0 \le i \le 2k-1\} \\ &\cup \{e_i \mid 1 \le i \le m\} \cup \{t, f\}, \\ T &= \{t\}, \\ H &= \{0, 1, 2, \ldots, m+1\}, \\ \mu &= [_{m+1}[_m \cdots [_1[_0]_0^0]_1^0 \cdots]_m^0]_{m+1}^0, \\ w_0 &= c_0 f,\ w_i = \lambda \text{ for } 1 \le i \le m+1. \end{aligned}$$

The rules are the following:

1. $[_0 c_i \to c_{i+1}]_0^\alpha$, for $0 \le i < 2k-1$ and $\alpha \in \{+, -, 0\}$,
   $[_0 c_{2k-1}]_0^\alpha \to t$, for $\alpha \in \{+, -, 0\}$
   (We count till $2k-1$, the time required to produce all k-combinations of nodes.)

2. $[_0 f]_0^0 \to [_0 a_1]_0^+ [_0 a_2]_0^- \cdots [_0 a_{n-k+1}]_0^\alpha$,
   where α is $+$ or $-$ depending on whether $n-k+1$ is odd or even.
   (We start the generation of all k-combinations by initially getting access to the first $n-k+1$ nodes.)

3. $[_0 a_i]_0^0 \rightarrow [_0 i a_{j_1}]_0^+ [_0 i a_{j_2}]_0^- \cdots [_0 i a_{j_l}]_0^\alpha$,
   for $j_1, j_2, \ldots, j_l > i$, $1 \leq i, j_1, \ldots, j_l \leq n$; α is $+$ or $-$ depending on whether l is odd or even.
   (We generate all k-combinations of the node set.)

4. $[_1 a_i \rightarrow i]_1^\alpha$, $\alpha \in \{+,-\}, 1 \leq i \leq n$.
   $[_1 a_i \rightarrow e_{j_1} \ldots e_{j_m}]_1^0$, if edges $e_{j_1}, \ldots, e_{j_m}$ are adjacent to node i.

5. $[_1 i \rightarrow e_{j_1} e_{j_2} \ldots e_{j_m}]_1^0$, if edges $e_{j_1}, e_{j_2}, \ldots, e_{j_m}$ are adjacent to node i.
   (Checking the coverage of edges by each k-combination. By listing the edges adjacent to each node, we can see if a particular k-combination covers all edges or not.)

6. $[_i e_i]_i^\alpha \rightarrow e_i$, for $1 \leq i \leq m, \alpha \in \{+,-,0\}$.
   (Membrane i is dissolved if edge e_i is covered.)

7. $[_{m+1} t]_{m+1}^0 \rightarrow [_{m+1}]_{m+1}^+ \mathrm{t}$.
   (The object t reaches the skin membrane only if all m edges are covered, that is, if all the m membranes in the substructure are dissolved. This means that at least one k-combination of nodes covers all m edges.)

8. $[_{i+1} [_i\ ]_i^+ [_i\ ]_i^- \cdots [_i\ ]_i^+]_{i+1}^0 \rightarrow [_{i+1} [_i\ ]_i^0]_{i+1}^+ [_{i+1} [_i\ ]_i^0]_{i+1}^- \cdots [_{i+1} [_i]_i^0]_{i+1}^+$, for $0 \leq i \leq m-2$
   $[_m [_{m-1}\ ]_{m-1}^+ [_{m-1}\ ]_{m-1}^- \cdots [_{m-1}\ ]_{m_1}^+]_m^0 \rightarrow [_m [_{m-1}\ ]_{m-1}^0]_m^0 [_m [_{m-1}\ ]_{m-1}^0]_m^0 \cdots [_m [_{m-1}]_{m-1}^0]_m^0$
   (Division rules for membranes labeled with $0, 1, \ldots, m$; the opposite polarization introduced when dividing a membrane 0 is propagated from lower levels to upper levels of the membrane structure and the membranes are continuously divided until dividing also membrane m – which will get neutral charge.)

Clearly, the system Π constructed above has n-bounded membrane division.

It can be easily seen that

$$L(\Pi) = \begin{cases} \{t\}, & \text{if there exists a node cover of size } k, \\ \emptyset, & \text{otherwise.} \end{cases}$$

Therefore, we get the answer to our decision problem by examining the emptiness of the language $L(\Pi)$. This can be done in $2k + m + 3$ steps: in $2k - 1$ steps, we generate all k-combinations of nodes. In the next step, membrane zero dissolves. In the next two steps, we list the adjacent edges corresponding to each node in all the k-combinations. In the next m steps, membranes 1 to m dissolve if edges e_1 to e_m are present in the expanded list of edges corresponding to the k-combinations. We then need one more step to output t, if any copy of t has reached the skin membrane. If no copy of t leaves the system at this step, then there is no node cover of size k for G. Note that we have used rules of type (a)-(f) except type (b).

6 Conclusion

We have introduced a new computability model, the P System based on parallel rewriting. Characterizations of *RE* languages are obtained by simple P Systems with one membrane. Of definite interest is to consider what happens if targets are nondeterministically chosen when a string is transformed; does this have any significant effect in increasing the power of the system .It is also of worthwhile interest to consider other NP-complete problems which can be solved using P Systems with d-bounded division.

Acknowledgement. The authors are much indebted to Gh. Păun, for many useful remarks about previous versions of this paper.

References

[1] J. Dassow, Gh. Păun, On the power of membrane computing, *J. of Universal Computer Sci.*, 5, 2 (1999), 33–49 (www.iicm.edu/jucs).

[2] S. N. Krishna, R. Rama, A variant of P systems with active membranes: Solving NP-complete problems, *Romanian J. of Information Science and Technology*, 2, 4 (1999).

[3] Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61(2000) and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).

[4] Gh. Păun, Computing with membranes – A variant: P Systems with Polarized Membranes, *Intern. J. of Foundations of Computer Science*, 11, 1 (2000), 167–182, and *Auckland University, CDMTCS Report* No 098, 1999 (www.cs.auckland.ac.nz/CDMTCS).

[5] Gh. Păun, P systems with active membranes: Attacking NP complete problems, submitted 1999, and *Auckland University, CDMTCS Report* No 102, 1999 (www.cs.auckland.ac.nz/CDMTCS).

[6] Gh. Păun, G. Rozenberg, A. Salomaa, Membrane computing with external output, *Fundamenta Informaticae*, 41, 3 (2000), 259–266.

[7] Gh. Păun, Y. Sakakibara, T. Yokomori, P systems on graphs of restricted forms, *IFIP Conf. on TCS: Exploring New Frontiers of Theoretical Informatics*, Sendai, Japan, 2000.

[8] Gh. Păun, T. Yokomori, Membrane computing based on splicing, *Preliminary Proc. of Fifth Intern. Meeting on DNA Based Computers* (E. Winfree, D. Gifford, eds.), MIT, June 1999, 213–227.

[9] Gh.Păun, S.Yu, On synchronization in P systems, *Fundamenta Informaticae*, 38, 4 (1999), 397–410.

[10] A.Salomaa, *Public Key Cryptography*, Springer-Verlag,Berlin, 1990.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 236 - 248.

Distributed Splicing of $\mathcal{RE}$ with 6 Test Tubes

Monika Sturm and Thomas Hinze

Dresden University of Technology, Germany
Department of Theoretical Computer Science
e-mail: {sturm,hinze}@tcs.inf.tu-dresden.de
www: http://wwwtcs.inf.tu-dresden.de/dnacomp

June 21, 2000

Abstract

This paper introduces a functional approach to distributed splicing systems for generation of recursive enumerable languages with 6 test tubes. The specification of this system serves both, the formal mathematical and the lab-experimental aspect. The implementation of the splicing system using a functional description of laboratory operations supports particularly the last-mentioned aspect. Advantages of this approach consist in large experimental practicability as well as in the independence of certain Chomsky type 0 grammar parameters.

1 Introduction

Fast solutions of combinatorial problems have both, large economical and theoretical importance. Starting from the consideration of different approaches to tackle NP-complete problems, several unconventional ideas for their solutions emerge. These ideas can be divided into four raw categories: neural, quantum, heuristic, and molecular computing, particularly using the data carrier DNA.

An interdisciplinary collaboration between computer scientists and molecular biologists developed a DNA based algorithm to solve the NP-complete knapsack problem with natural object weights and implemented it repetitively in the laboratory. In parallel to these experimental studies the laboratory-like DNA computing model DNA-Haskell[13] was developed based on biochemical processes observed in detail including some side effects that can occur indeed. DNA computing models feature by their computational completeness. DNA-Haskell also owns this property proved by simulation of selected conventional universal models for computation.

Some DNA computing models are characterized by a high abstraction level and by a clear formal model specification. Between these models and an according implementation in the laboratory a gap exists that has to be discussed. Simulating those DNA computing models using the laboratory-like DNA-Haskell could be a promising approach to fill this gap and to combine the advantages of different models. The minimization of the number of used test tubes necessary for the execution of DNA based algorithms acts as a significant criterion to optimize DNA computing models. A one pot architecture embodies the ideal case. The splicing model is closed to this ideal. The splicing operation forms the principal item of this model. It is possible to simulate the splicing operation in DNA-HASKELL. Splicing systems were established to reach universal computational power[4]. They require either an infinite set of axioms or an infinite set of splicing rules to generate recursively enumerable languages (class $\mathcal{RE}$). A further approach based on multisets leads to the necessary to determine the number of strand duplicates with high accuracy. The recent state of the art in molecular bioengineering can not meet these requirements completely. Therefore other extensions of splicing systems were sought for a practicable possibility to generate $\mathcal{RE}$. The introduction of distributed splicing systems with n test tubes seems to be a successful way. A functional approach to distributed splicing of $\mathcal{RE}$ with 6 test tubes is proposed below. The arisen distributed splicing system, named TT6 for short, owns a simulation in DNA-HASKELL and performs a distribution of test tube contents after each splicing operation. Further the TT6 uses compact sets of filter patterns resulting in a comparable low exchange of DNA strands between test tubes supporting a lab-implementation.

2 On the Basic Operation in Splicing Systems

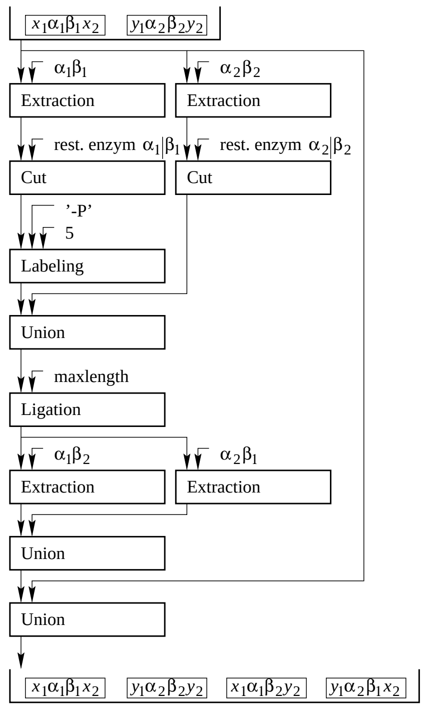
The splicing operation forms the core of all types of splicing systems and embodies an abstract formal emulation of DNA recombinant techniques cut with restriction enzymes (digestion) and ligation [4]. It is based on elements of mostly infinite sets that express DNA strands, further named words of formal languages. The description of the splicing operation on words of formal languages also leads to a generalization of the effect that is caused by digestion and ligation. The generalization suppresses certain DNA strands resp. words that can really additional occur during the ligation process as side effects. Here, we propose a sequence of DNA-HASKELL operations that simulate the splicing operation on linear data structures defined by a splicing rule in [6].

Consider an alphabet Σ, and two symbols \$ and # not in Σ. A splicing rule over Σ is a string $r = \alpha_1\#\beta_1\$\alpha_2\#\beta_2$, where $\alpha_i, \beta_i \in \Sigma^*$, $1 \leq i \leq 2$.
For each such rule r and strings $x, y, w, z \in \Sigma^*$ we define

$$(x, y) \vdash_r (z, w) \text{ if and only if } \quad \begin{array}{ll} x = x_1\alpha_1\beta_1x_2, & y = y_1\alpha_2\beta_2y_2, \\ z = x_1\alpha_1\beta_2y_2, & w = y_1\alpha_2\beta_1x_2. \end{array}$$

A wet splicing system and its experimental implementation was introduced in [8]. The results of this approach encourage the assumption that the splicing operation can be performed practically. This fact represents the first step to establish a universal DNA computer based on splicing.

Unfortunally the ligation produces unwanted DNA fragments (e.g. of the composition $x_1\alpha_1\alpha_2y_1$ and $y_2\beta_2\beta_1x_2$). These DNA fragments occur because of the compatibility of the sticky ends resulting from the digestions $\alpha_1|\beta_1$ and $\alpha_2|\beta_2$. These additional unwanted DNA fragments can hardly influence the final result of iterated executions of splicing operations, often used to generate a formal language by applying splicing rules of a splicing system. The following figure 1 proposes an idea how to overcome this insufficiency. Let x and y be encoded by DNA double strands.



```
xy_pool :: Tube
α1β1 :: Dnastrand
α2β2 :: Dnastrand
α1β2 :: Dnastrand
α2β1 :: Dnastrand
maxlength :: Int

combine :: Tube -> Tube
combine t = lig
              (un
                (lab
                  (cut
                    (extr t α1β1)
                    'restriction enzym α1|β1')
                  '-P'
                  5)
                (cut
                  (extr t α2β2)
                  'restriction enzym α2|β2'))
              maxlength

splicingop :: Tube -> Tube
splicingop t = un
                 (un
                   (extr
                     (combine t)
                     α1β2)
                   (extr
                     (combine t)
                     α2β1))
                 t
```

application of the function:

```
splicingop xy_pool
```

Figure 1: splicing operation as a flowchart (left) and using the DNA-HASKELL syntax (right)

We are able to describe the splicing operation in a experimental convincing way. DNA computing should lead to a unconventional universal model for computations. Splicing systems based on the splicing operation represent an exact mathematical model according to this aim [7]. A practical execution of different splicing systems can contain the scenario from figure 2 as the central part. The challenge consists in finding a way how to adapt the real molecular biological processes in the laboratory to the formal definition of the splicing operation. The focus lies in the laboratory-like modelling of splicing systems to generate regular, context free, and recursive enumerable languages.

3 Splicing Systems for $\mathcal{RE}$ – Comparison and Classification

The sets $\mathcal{FIN}$, $\mathcal{REG}$, $\mathcal{CF}$, $\mathcal{CS}$, $\mathcal{RE}$ are used to denote the classes of finite, regular (Chomsky type 3), context free (2), context sensitive (1), and recursive enumerable (0) languages. These classes form the Chomsky hierarchy.

To describe and characterize these classes of languages, different formal systems like Chomsky grammars of a certain type were developed. Each of these language denotation systems is able to generate exactly those words the language is composed of. Language denotation systems represent in general models for computation: Since a language can be considered as a set (finite or infinite) consisting of its words as elements, generating a language by producing its words and testing whether or not a given string is a word of the described language can be assumed as computational process. From $\mathcal{FIN}$ to $\mathcal{RE}$ the computational power increases and descriptions of $\mathcal{RE}$ are said to be computational universal. Therefore, the generation of $\mathcal{RE}$ by splicing systems based on an appropriate language denotation system will be focussed.

Some extended splicing systems $EH(\mathcal{F}_1, \mathcal{F}_2)$, $\mathcal{F}_1$: set of axioms, $\mathcal{F}_2$: set of splicing rules, can produce $\mathcal{RE}$, see table 1 [4]:

$\mathcal{F}_1/\mathcal{F}_2$	$\mathcal{FIN}$	$\mathcal{REG}$	$\mathcal{CF}$	$\mathcal{CS}$	$\mathcal{RE}$
$\mathcal{FIN}$	$\mathcal{REG}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$
$\mathcal{REG}$	$\mathcal{REG}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$
$\mathcal{CF}$	$\mathcal{CF}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$
$\mathcal{CS}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$
$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$	$\mathcal{RE}$

Table 1: computational power of EH systems

Weakest preconditions are required by $EH(\mathcal{FIN}, \mathcal{REG})$ and $EH(\mathcal{CS}, \mathcal{FIN})$. These systems need either a regular language to describe the splicing rules or a context sensitive language to describe the axioms. Both, $\mathcal{REG}$ and $\mathcal{CS}$, are infinite sets resulting in an infinite number of DNA strands and/or restriction enzymes for simulating the work of these splicing systems. To overcome this insufficiency, three different ideas leading to special types of extended splicing systems were pursued:

- introduction of multisets
- extension of the basic data structure "finite linear string"
- introduction of distributed splicing systems usually with more than one test tube and control mechanisms for the resulting test tube systems

Table 2 illustrates extended splicing systems able to generate the class of recursive enumerable languages together with a short characterization of some properties. Further considerations focus on classes $\mathcal{FIN}$, $\mathcal{REG}$, and $\mathcal{RE}$.

class	splicing system	reference	number of test tubes	number of axioms	number of splicing rules	basic data structure	required language denotation	supports multiple instructions
distributed	Multiple	[12]	1	$O(\|\mathcal{M}\|)$	$O(\|\mathcal{D}\|^3)$	linear	Elementary Formal System (EFS) $EFS = (\mathcal{D}, \Sigma, \mathcal{M})$, Post Normal System (PNS) $G = (\mathcal{V}, \Sigma, \mathcal{P}, \mathcal{A})$	no
	TT system	[2]	$O(\|\Sigma_G\|)$	$O(\|\mathcal{P}_G\| + \|\Sigma_G\|)$	$O(\|\mathcal{P}_G\| + \|\Sigma_G\|)$	linear	Chomsky type 0 grammar $G = (\mathcal{V}_G, \Sigma_G, \mathcal{P}_G, S_G)$	yes
	CDEH system	[9] [11]	3	$O(\|\mathcal{P}_G\| + \|\mathcal{V}_G\| + \|\Sigma_G\|)$	$O(\|\mathcal{P}_G\| + \|\mathcal{V}_G\| + \|\Sigma_G\|)$	linear	Chomsky type 0 grammar $G = (\mathcal{V}_G, \Sigma_G, \mathcal{P}_G, S_G)$	yes
	TT6 system	here	6	$O(\|\mathcal{P}_G\| + \|\Sigma_G\|)$	$O(\|\mathcal{P}_G\| + \|\Sigma_G\|)$	linear	Chomsky type 0 grammar $G = (\mathcal{V}_G, \Sigma_G, \mathcal{P}_G, S_G)$	yes
infinite set	$EH(\mathcal{FIN}, \mathcal{REG})$	[10]	1	$O(\|\mathcal{P}_G\| + \|\mathcal{V}_G\| + \|\Sigma_G\|)$	∞	linear	Chomsky type 0 grammar $G = (\mathcal{V}_G, \Sigma_G, \mathcal{P}_G, S_G)$	no
extended data structure	Circular	[14]	1	$O(\|\mathcal{P}\| + \|\mathcal{A}\|)$	$O(\|\mathcal{P}\|)$	circular	Post Normal System $G = (\mathcal{V}, \Sigma, \mathcal{P}, \mathcal{A})$	no
multiset	$EH(m\mathcal{FIN}, \mathcal{FIN})$	[1]	1	$O(\|\mathcal{P}_G\| + \|\mathcal{V}_G\| + \|\Sigma_G\|)$	$O(\|\mathcal{P}_G\| \cdot (\|\mathcal{V}_G\| + \|\Sigma_G\|)^5)$	linear	Chomsky type 0 grammar $G = (\mathcal{V}_G, \Sigma_G, \mathcal{P}_G, S_G)$	no
		[5]	1	$O(\|\mathcal{V}\|^2\|\mathcal{Q}\| + \|\mathcal{Q}\|^2\|\mathcal{V}\|)$	$O(\|\mathcal{V}\|^4\|\mathcal{Q}\|)$	linear	deterministic Turing Machine $TM = (\mathcal{Q}, \mathcal{V}, \Sigma, \{L, R\}, \mathcal{F}, q_0, \varepsilon, \delta)$	no
		[3]	1	$O(\|\Sigma\| \cdot \|\mathcal{P}\|)$	$O(\|\Sigma\| \cdot \|\mathcal{P}\|)$	linear	Post Normal System $G = (\mathcal{V}, \Sigma, \mathcal{P}, \mathcal{A})$	no

Table 2: classification of splicing systems able to generate recursive enumerable languages

Table 2 shows that TT6 as a model for distributed splicing is characterized by a linear complexity of the number of axioms and splicing rules depending on the number of grammar rules and terminal symbols of the language. Further the TT6 uses 6 test tubes in any case working on a linear DNA data structure.

4 A Test Tube 6 Distributing Splicing System Γ for Chomsky Type 0 Grammars

The definition of Γ uses the components of Chomsky type 0 grammars that have to be given for the construction of a concrete Γ.
Let $G = (\mathcal{V}_G, \Sigma_G, \mathcal{P}_G, S_G)$ be a Chomsky type 0 grammar with

$\mathcal{V}_G$:	finite set of nonterminal symbols			
Σ_G:	finite set of terminal symbols	Σ_G	$=$	$\{\sigma_1, \sigma_2, \ldots, \sigma_n\}, \quad \mathcal{V}_G \cap \Sigma_G = \emptyset$
$\mathcal{P}_G$:	finite set of productions	$\mathcal{P}_G$	$\subset$	$((\mathcal{V}_G \cup \Sigma_G)^* \otimes \mathcal{V}_G \otimes (\mathcal{V}_G \cup \Sigma_G)^*) \times ((\mathcal{V}_G \cup \Sigma_G)^+)$
S_G:	start symbol	S_G	$\in$	$\mathcal{V}_G$

Let $\Gamma = (\mathcal{V}, T_1, T_2, T_3, T_4, T_5, T_6)$ be a Test Tube Distributed Extended Head System of degree 6 (TT6 for short) based on G with

$\mathcal{V}$:	finite set of alphabet symbols	$\mathcal{V}$	$=$	$\mathcal{V}_G \cup \Sigma_G \cup \{B, \alpha, \beta, X, X', Y, Y', Y'_\alpha, Y'_\beta, Z, Z', Z''\}$
T_i:	test tube i, $i = 1, \ldots, 6$ with	T_i	$=$	$(\mathcal{A}_i, \mathcal{R}_i, \mathcal{F}_i)$
$\mathcal{A}_i$:	finite set of axioms	$\mathcal{A}_i$	$\subset$	$\mathcal{V}^*$
$\mathcal{R}_i$:	finite set of splicing rules	$\mathcal{R}_i$	$\subset$	$\mathcal{V}^* \otimes \{\#\} \otimes \mathcal{V}^* \otimes \{\$\} \otimes \mathcal{V}^* \otimes \{\#\} \otimes \mathcal{V}^*$
$\mathcal{F}_i$:	finite set of filter patterns	$\mathcal{F}_i$	$\subset$	$(\mathcal{V} \cup \{@\})^*$, with an ambiguous letter @, $\{@\}^* \equiv \{@\}$, that stands for any finite word $\in \mathcal{V}^*$
ε:	empty word			

Each test tube T_i is called a *component* of Γ. Any $\mathcal{A}_i$ can be represented as finite languages over $\mathcal{V}$, and any $\mathcal{R}_i$ can be represented as finite languages over $\mathcal{V} \cup \{\#\} \cup \{\$\}$. # and \$ are auxiliary symbols not in $\mathcal{V}$. For simplicity, the operation $^+$ on an arbitrary set $\mathcal{U}$ is defined as $\mathcal{U}^+ := \mathcal{U}^* \otimes \mathcal{U}$. It excludes ε from the result of the *-operation. The components of Γ are defined as follows:

$$T_1 = (\mathcal{A}_1, \mathcal{R}_1, \mathcal{F}_1)$$
$$\mathcal{A}_1 = \{XBS_GY\} \cup \{ZvY' \mid \exists u \in (\mathcal{V}_G \cup \Sigma_G)^+.(u,v) \in \mathcal{P}_G\}$$
$$\mathcal{R}_1 = \{r_{11}\}$$
$$r_{11} = \varepsilon\#uY\$Z\#vY'; \;\; (u,v) \in \mathcal{P}_G$$
$$\mathcal{F}_1 = \{@Y\}$$

$$
\begin{array}{rcl}
T_2 & = & (\mathcal{A}_2, \mathcal{R}_2, \mathcal{F}_2) \\
& & \mathcal{A}_2 \;=\; \{Z\beta\alpha^i\beta Y' \mid i = 1, \ldots, n\} \cup \{ZY'_\alpha, ZY'_\beta, X'Z, ZY\} \\
& & \mathcal{R}_2 \;=\; \{r_{21}, r_{22}, r_{23}, r_{24}, r_{25}, r_{26}\} \\
& & \qquad r_{21} \;=\; \varepsilon\#\sigma_i Y'\$Z\#\beta\alpha^i\beta Y';\; i = 1, \ldots, n \\
& & \qquad r_{22} \;=\; \varepsilon\#\beta Y'\$Z\#Y'_\beta \\
& & \qquad r_{23} \;=\; \varepsilon\#\alpha Y'\$Z\#Y'_\alpha \\
& & \qquad r_{24} \;=\; X\#\varepsilon\$X'\#Z \\
& & \qquad r_{25} \;=\; X\beta\alpha^i\beta\#\varepsilon\$X\sigma_i\#Z;\; i = 1, \ldots, n \\
& & \qquad r_{26} \;=\; \varepsilon\#Y'\$Z\#Y \\
& & \mathcal{F}_2 \;=\; \{X@\sigma_i Y' \mid i = 1, \ldots, n\} \cup \{X@Y'_\alpha, X@Y'_\beta, X\beta@, X\alpha@\}
\end{array}
$$

$$
\begin{array}{rcl}
T_3 & = & (\mathcal{A}_3, \mathcal{R}_3, \mathcal{F}_3) \\
& & \mathcal{A}_3 \;=\; \{ZY', X\alpha Z\} \\
& & \mathcal{R}_3 \;=\; \{r_{31}, r_{32}\} \\
& & \qquad r_{31} \;=\; \varepsilon\#Y'_\alpha\$Z\#Y' \\
& & \qquad r_{32} \;=\; X'\#\varepsilon\$X\alpha\#Z \\
& & \mathcal{F}_3 \;=\; \{X'@Y'_\alpha\}
\end{array}
$$

$$
\begin{array}{rcl}
T_4 & = & (\mathcal{A}_4, \mathcal{R}_4, \mathcal{F}_4) \\
& & \mathcal{A}_4 \;=\; \{ZY', X\beta Z\} \\
& & \mathcal{R}_4 \;=\; \{r_{41}, r_{42}\} \\
& & \qquad r_{41} \;=\; \varepsilon\#Y'_\beta\$Z\#Y' \\
& & \qquad r_{42} \;=\; X'\#\varepsilon\$X\beta\#Z \\
& & \mathcal{F}_4 \;=\; \{X'@Y'_\beta\}
\end{array}
$$

$$
\begin{array}{rcl}
T_5 & = & (\mathcal{A}_5, \mathcal{R}_5, \mathcal{F}_5) \\
& & \mathcal{A}_5 \;=\; \{Z'Z', Z''Z''\} \\
& & \mathcal{R}_5 \;=\; \{r_{51}, r_{52}\} \\
& & \qquad r_{51} \;=\; \varepsilon\#BY\$Z'Z'\#\varepsilon \\
& & \qquad r_{52} \;=\; X\#\varepsilon\$\varepsilon\#Z''Z'' \\
& & \mathcal{F}_5 \;=\; \{@BY\}
\end{array}
$$

$$
\begin{array}{rcl}
T_6 & = & (\mathcal{A}_6, \mathcal{R}_6, \mathcal{F}_6) \\
& & \mathcal{A}_6 \;=\; \emptyset \\
& & \mathcal{R}_6 \;=\; \emptyset \\
& & \mathcal{F}_6 \;=\; \Sigma_G^+
\end{array}
$$

The work of Γ can be described as an iterated loop in each of the test tubes T_i, $i = 1, \ldots, 5$. The iterated loop consists of the consecutive executed steps splicing, filtering, and distributing. The test tube T_6 has the function of a final tube and stores the resulting words $\in \mathcal{L}(G)$. The current contents of test tube T_i is denoted as $\mathcal{L}_i$. Initially, $\mathcal{L}_i = \mathcal{A}_i$ with $i = 1, \ldots, 6$.

The iterated loop in each test tube T_i, $i = 1, \ldots, 5$ starts with the according sets of axioms $\mathcal{A}_i$. Subsequently, any executable splicing rule has been selected and applied. If there is no applicable splicing rule, no splicing will be performed and the set of strings in the relevant test tube remains unchanged. The application of a splicing operation in a test tube T_i with the contents $\mathcal{L}_i \subset \mathcal{V}^*$ is defined by the function σ:

$$\sigma(\mathcal{L}_i) := \{z \in \mathcal{V}^* \mid (x,y) \vdash_r (z,w) \; or \; (x,y) \vdash_r (w,z), \; for \; some \; x,y \in \mathcal{L}_i, r \in \mathcal{R}_i\}$$

During the splicing step of the iterated loop the splicing operation is performed at most once in T_i. In consequence, the k-fold iterated splicing $\sigma^k(\mathcal{L}_i)$ is defined by

$$\begin{array}{rcl} \sigma^0(\mathcal{L}_i) & = & \mathcal{L}_i \\ \sigma^{k+1}(\mathcal{L}_i) & = & \sigma^k(\mathcal{L}_i) \cup \sigma(\sigma^k(\mathcal{L}_i)) \; for \; k \geq 0, \; 1 \leq i \leq 5 \end{array}$$

with

$$\sigma^*(\mathcal{L}_i) = \bigcup_{k \geq 0} \sigma^k(\mathcal{L}_i).$$

During one pass of the iterated loop the splicing operation is executed at most once in each test tube T_1 until T_5.

After splicing, the filtering step prepares copies of those strings that have to be distributed. Every test tube T_i, $i = 1, \ldots, 5$ provides separately exactly those strings that will be moved into other test tubes. To do so, T_i evaluates all filter pattern $\mathcal{F}_j$, $j = 1, \ldots, 5$, $j \neq i$. Those strands that are transmitted into other tubes are removed from the producing tube T_i if they do not match its own filter pattern $\mathcal{F}_i$. Each $\mathcal{F}_i$ describes those strings that are moved from other test tubes into T_i.

The subsequent distributing step exchanges the strings prepared by filtering between the test tubes.

The test tube T_6 receives copies of all strings produced by splicing in T_1 until T_5 during each iterated loop and collects those strings that describe words of the language $\mathcal{L}(G)$, implemented by its filter pattern $\mathcal{F}_6$. All other arriving strings are eliminated. T_6 does not practise a splicing operation itself, it evaluates the produced words from all other test tubes. Because of this behavior T_6 carries the meaning of a final tube (master tube).

The steps splicing, filtering, and distributing forming the iterated loop are executed consecutively. After distributing, the next round of splicing starts. All test tubes T_1 until T_5 perform iterated loops in parallel. The number of iterated loops is not limited by Γ. One pass of the iterated loop transforms stepwise the contents $(\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_6)$ into $(\mathcal{L}'_1, \mathcal{L}'_2, \ldots, \mathcal{L}'_6)$, denoted by the $\xrightarrow{1}_{TT6}$ operator

$$(\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_6) \xrightarrow{1}_{TT6} (\mathcal{L}'_1, \mathcal{L}'_2, \ldots, \mathcal{L}'_6)$$

with

$$\mathcal{L}'_i = \left(\bigcup_{\substack{j=1 \wedge \\ j \neq i}}^{6} \sigma^k(\mathcal{L}_j) \right) \cap \mathcal{F}_i \cup \left(\sigma^k(\mathcal{L}_i) \setminus \left(\bigcup_{\substack{j=1 \wedge \\ j \neq i}}^{6} \mathcal{F}_j \right) \right), \; k \in \{0,1\}, \; 1 \leq i \leq 6.$$

The contents of each T_i is composed by the result of the own splicing minus those strings that move to other test tubes plus those strings that arrive from other test tubes.

Therefore, the language $\mathcal{L}$ generated by Γ is described by

$$\mathcal{L}(\Gamma) = \left\{ w \in \Sigma_G^+ \ \wedge \ w \in \mathcal{L}_6 \mid (\mathcal{A}_1, \ldots, \mathcal{A}_6) \xrightarrow{*}_{TT6} (\mathcal{L}_1, \ldots, \mathcal{L}_6) \right\},$$

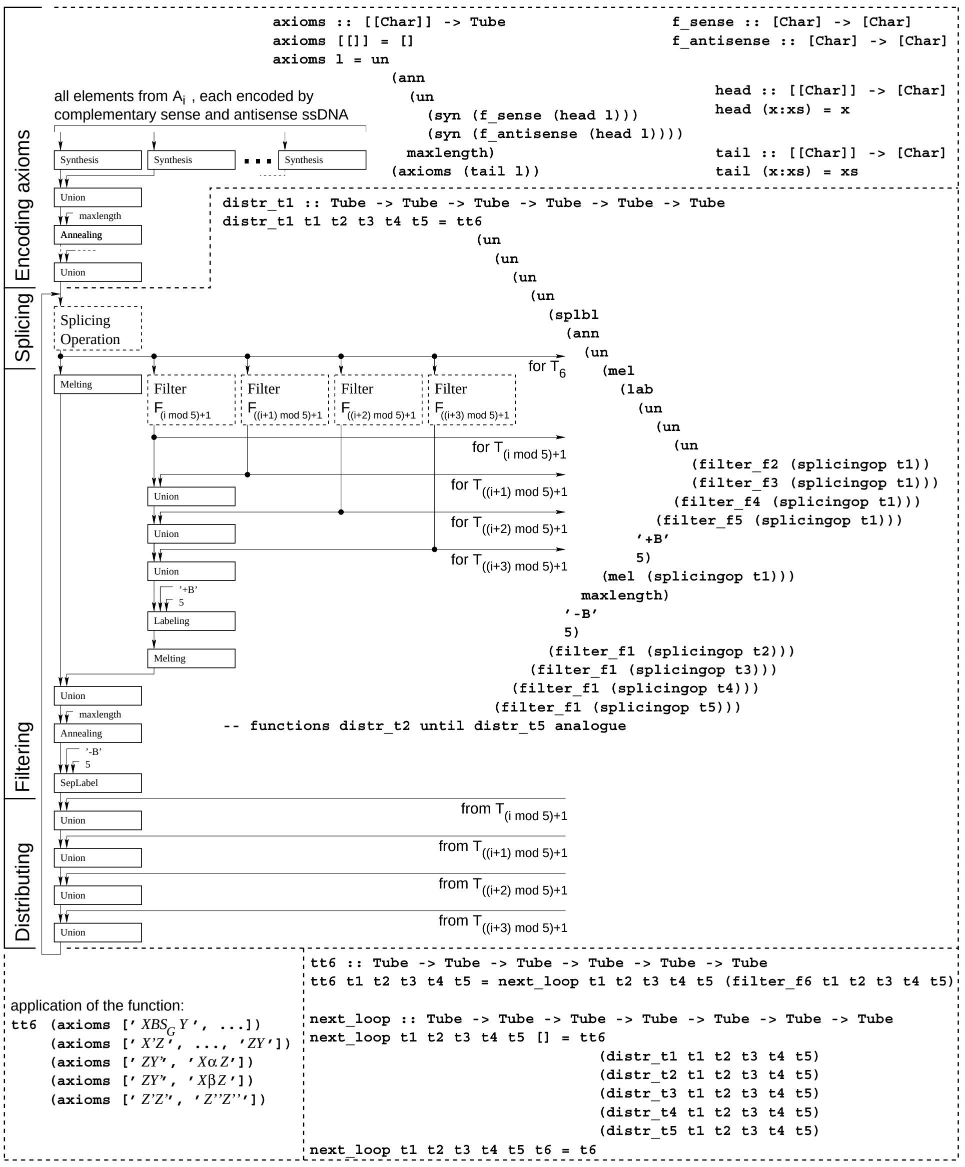
where $\xrightarrow{*}_{TT6}$ is the reflexive and transitive closure of $\xrightarrow{1}_{TT6}$.

Each of the splicing test tubes assumes a special task in the whole behavior of Γ, listed in table 3.

test tube	**task**
T_1	• apply a production of the grammar G
T_2	• encode the rightmost completing terminal symbol $\sigma_j \in \Sigma_G$ into the sequence $\beta\alpha^j\beta$ if existing • prepare the rightmost-leftmost rotation of the completing α or β • decode the leftmost complete rotated $\beta\alpha^j\beta$ into the terminal symbol σ_j
T_3	• rightmost-leftmost rotation of one symbol α
T_4	• rightmost-leftmost rotation of one symbol β
T_5	• extract a ready generated word of the language $\mathcal{L}(G)$ from the terminating auxiliary symbols X and BY

Table 3: tasks of the splicing test tubes

Figure 2 shows the general iterated loop practised in T_1 until T_5. Each iterated loop in T_i, $i = 1, \ldots, 5$ has been initialized with the axioms from the according set $\mathcal{A}_i$ encoded by appropriate unique DNA double strands. The encoding is done by two one to one functions `f_sense` and `f_antisense` that produce complementary single stranded DNA strings from any $\in V^+$. The body of each iterated loop is composed by steps splicing, filtering, and distributing. Each splicing step executes the splicing operation according to $\mathcal{R}_i$ at most once. The subsequent filtering step prepares in each T_i copies of those DNA strands according to filter patterns $\mathcal{F}_j$, $j \neq i$ separately. All prepared DNA strands matching $\mathcal{F}_j$ are distributed to T_j. $\mathcal{F}_6$ extracts those DNA strands representing an arbitrary word $\in \mathcal{L}(G)$ and collects it in T_6. The iterated loops of TT6 terminate as soon as an arbitrary word $\in \mathcal{L}(G)$ exists in T_6. The case $\mathcal{L}(G) = \emptyset$ leads to a nonterminating process. This consequence coincides to a terminating property of programs with mutual recursion.



Encoding axioms
Splicing
Filtering
Distributing
all elements from A_i , each encoded by complementary sense and antisense ssDNA
Synthesis
Synthesis
Synthesis
Union
maxlength
Annealing
Union
Splicing Operation
Melting
Filter $F_{(i \bmod 5)+1}$
Filter $F_{((i+1) \bmod 5)+1}$
Filter $F_{((i+2) \bmod 5)+1}$
Filter $F_{((i+3) \bmod 5)+1}$
for T_6
for $T_{(i \bmod 5)+1}$
for $T_{((i+1) \bmod 5)+1}$
for $T_{((i+2) \bmod 5)+1}$
for $T_{((i+3) \bmod 5)+1}$
Union
Union
Union
'+B'
5
Labeling
Melting
Union
maxlength
Annealing
'-B'
5
SepLabel
Union
Union
Union
Union
from $T_{(i \bmod 5)+1}$
from $T_{((i+1) \bmod 5)+1}$
from $T_{((i+2) \bmod 5)+1}$
from $T_{((i+3) \bmod 5)+1}$

```
axioms :: [[Char]] -> Tube                     f_sense :: [Char] -> [Char]
axioms [[]] = []                               f_antisense :: [Char] -> [Char]
axioms l = un
                (ann
                  (un                               head :: [[Char]] -> [Char]
                    (syn (f_sense (head l)))        head (x:xs) = x
                    (syn (f_antisense (head l))))
                  maxlength)                        tail :: [[Char]] -> [Char]
                (axioms (tail l))                   tail (x:xs) = xs
```

```
distr_t1 :: Tube -> Tube -> Tube -> Tube -> Tube -> Tube
distr_t1 t1 t2 t3 t4 t5 = tt6
                              (un
                                (un
                                  (un
                                    (un
                                      (splbl
                                        (ann
                                          (un
                                            (mel
                                              (lab
                                                (un
                                                  (un
                                                    (un
                                                      (filter_f2 (splicingop t1))
                                                      (filter_f3 (splicingop t1)))
                                                    (filter_f4 (splicingop t1)))
                                                  (filter_f5 (splicingop t1)))
                                                '+B'
                                                5)
                                            (mel (splicingop t1)))
                                          maxlength)
                                        '-B'
                                        5)
                                      (filter_f1 (splicingop t2)))
                                    (filter_f1 (splicingop t3)))
                                  (filter_f1 (splicingop t4)))
                                (filter_f1 (splicingop t5)))
-- functions distr_t2 until distr_t5 analogue
```

application of the function:

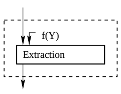
```
tt6 (axioms ['XBS_G Y', ...])
    (axioms ['X'Z', ..., 'ZY'])
    (axioms ['ZY'', 'XαZ'])
    (axioms ['ZY'', 'XβZ'])
    (axioms ['Z'Z'', 'Z''Z''''])
```

```
tt6 :: Tube -> Tube -> Tube -> Tube -> Tube -> Tube
tt6 t1 t2 t3 t4 t5 = next_loop t1 t2 t3 t4 t5 (filter_f6 t1 t2 t3 t4 t5)

next_loop :: Tube -> Tube -> Tube -> Tube -> Tube -> Tube -> Tube
next_loop t1 t2 t3 t4 t5 [] = tt6
                                  (distr_t1 t1 t2 t3 t4 t5)
                                  (distr_t2 t1 t2 t3 t4 t5)
                                  (distr_t3 t1 t2 t3 t4 t5)
                                  (distr_t4 t1 t2 t3 t4 t5)
                                  (distr_t5 t1 t2 t3 t4 t5)
next_loop t1 t2 t3 t4 t5 t6 = t6
```

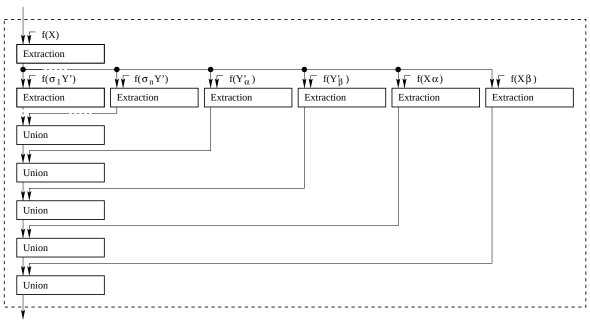
Figure 2: iterated loop for T_i, $i = 1, \ldots, 5$; flowchart and DNA-HASKELL syntax [13]

$F_1 = \{@Y\}$



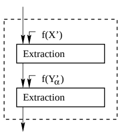
```
filter_f1 :: Tube -> Tube
filter_f1 t = extr t (f Y)
```

$F_2 = \{X@\sigma_i Y', X@Y'_\alpha, X@Y'_\beta, X\beta@, X\alpha@ \mid i=1,\dots,n\}$



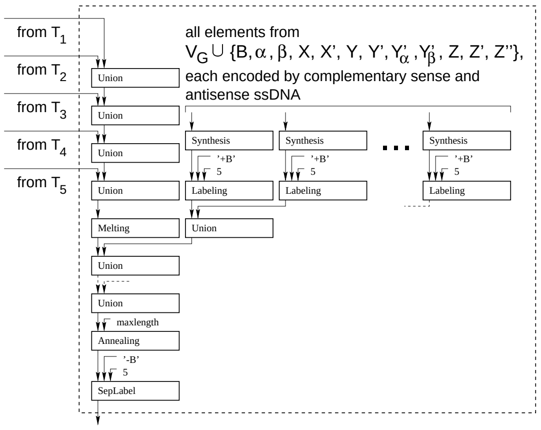
```
filter_f2 :: Tube -> Tube
filter_f2 t = un
                ...
                  (un
                    (un
                      (un
                        (un
                          (extr
                            (extr t (f X))
                            (f Y’α))
                          (extr
                            (extr t (f X))
                            (f Y’β))
                        (extr t (f Xβ)))
                      (extr t (f Xα)))
                    (extr
                      (extr t (f X))
                      (f σ1Y’)))
                  ...
                (extr
                   (extr t (f X))
                   (f σnY’))
```

$F_3 = \{X'@Y'_\alpha\}$



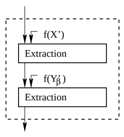
```
filter_f3 :: Tube -> Tube
filter_f3 t = extr
                (extr t (f X’))
                (f Y’α)
```

$F_6 = \{\sigma_i \mid i=1,\dots,n\}^+$



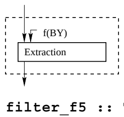
```
filter_f6 :: Tube -> Tube -> Tube -> Tube -> Tube -> Tube
filter_f6 t1 t2 t3 t4 t5 = splbl
                             (ann
                               (un
                                 (un
                                   ...
                                     (un
                                       (lab (syn (f_antisense B)) ’+B’ 5)
                                       (lab (syn (f_sense B)) ’+B’ 5))
                                     ...
                                   (lab (syn (f_sense Z’’)) ’+B’ 5))
                                 (mel
                                   (un
                                     (un
                                       (un
                                         (un t1 t2)
                                         t3)
                                       t4)
                                     t5)))
                               maxlength)
                             ’-B’
                             5
```

$F_4 = \{X'@Y'_\beta\}$



```
filter_f4 :: Tube -> Tube
filter_f4 t = extr
                (extr t (f X’))
                (f Y’β)
```

$F_5 = \{@BY\}$



```
filter_f5 :: Tube -> Tube
filter_f5 t = extr t (f BY)
```

Figure 3: filter patterns $\mathcal{F}_1$ until $\mathcal{F}_6$ and implementation of filtering processes including collection of words $\in \mathcal{L}(G)$ in T_6; flowchart and DNA-Haskell syntax

5 Conclusions

This paper implies a proposal to the discussion about distributed splicing systems. The objectives leading to the development of TT6 include the compliance with $\mathcal{RE}$ by a constant number of test tubes (6), by a nonextended DNA structure, and by an efficient derivation of complexity theoretical relevant system parameters directly from the grammar. Let Σ_G be the set of terminal symbols and $\mathcal{P}_G$ the set of grammar rules, then TT6 requires $O(|\mathcal{P}_G| + |\Sigma_G|)$ axioms and splicing rules independent of the number of nonterminal grammar symbols.

TT6 is constructed with regard to a practicable implementation in the laboratory. The distribution of DNA strands between test tubes is organized in a way that minimizes the number of transferred DNA strands. Beyond only few strand duplicates are necessary to perform all filtering and distributing processes. The number of DNA double strands that have to be available initially is equal to the number of axioms. The operations defined in the specification of DNA-HASKELL are based on observable processes in the laboratory. The experimental practicability of each single operation was shown.

References

[1] E. Csuhaj-Varjú, R. Freund, L. Kari, G. Paun. DNA computation based on splicing: universality results. Technical report 185-2/FR-2/95, TU Wien, Institute for Computer Languages, Wien, Austria, 1995

[2] E. Csuhaj-Varjú, L. Kari, G. Paun. Test tube distributed systems based on splicing. Computers and AI, vol. 15(2–3), p. 211-232, 1996

[3] C. Ferretti, G. Mauri, S. Kobayashi, T. Yokomori. On the Universality of Post and Splicing Systems, University of Electro-Communications, Chofu, Japan, 1998

[4] R. Freund, L. Kari, G. Paun. DNA computing based on splicing: the existence of universal computers. Theory of Computing Systems, vol. 32, p. 69-112, 1999

[5] P. Frisco, C. Mauri, C. Ferretti. Simulating Turing machines through extended mH systems, Università di Milano, Dipartimento di Scienze dell'Informazione, 1998

[6] T. Head. Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. Bulletin of the Mathematical Biology, vol. 49(6), p. 737-759, 1987

[7] L. Kari. DNA computing: the arrival of biological mathematics. The mathematical Intelligencer, vol. 19, 2, 1997

[8] E. Laun, K. J. Reddy. Wet splicing systems. In Proceedings of the 3rd DIMACS Workshop on DNA Based Computers, University of Pennsylvania, p. 115-126, 1997

[9] C. Martin-Vide, G. Paun. Cooperating distributed splicing systems. Workshop on molecular computing, 1997

[10] G. Paun. SPLICING – a challenge for formal language theorists. Journal of Automata, Languages and Combinatorics, vol. 4, no. 1, p. 3-16, 1999

[11] G. Paun, G. Rozenberg, A. Salomaa. DNA Computing. New Computing Paradigms. Springer, 1998

[12] Y. Sakakibara. Splicing, Tree Splicing, and Multiple Splicing. Tokyo Denki University, Department of Information Sciences, 1997

[13] E. Stoschek, M. Sturm, T. Hinze. On a DNA experiment for solving a certain NP-complete problem. Technical Report TUD-FI99-02, Dresden University of Technology, 1999

[14] T. Yokomori, S. Kobayashi, C. Ferretti. On the power of circular splicing systems and DNA computability. IEEE International Conference on Evolutionary Computation, Indiana University, Purdue University, Indianapolis, Illinois, 1997

[15] C. Zandron, C. Ferretti, G. Mauri. A reduced distributed splicing system for RE languages. In Gheorghe Paun and Arto Salomaa, editors, Lecture Notes in Computer Science, vol. 1218, p. 346-366. Springer Verlag, Berlin, Heidelberg, New York, 1997

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 249 - 264.

Core Memory Objects with Address Registers Representing High-dimensional Interaction

Hideaki Suzuki

ATR International, Information Sciences Division

2-2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0288 Japan

hsuzuki@isd.atr.co.jp

Abstract

When we design a computational system using a comparison between information objects and chemical molecules, we have to make the objects interact with each others tightly to make them create higher functions like biological catalyses. The one-dimensional core memory programming system SeMar is revised on this notion, in which an object is represented by a variable-length string and object-object interactions are represented by a set of address registers attached to an object. The register set is manipulated by a sequence of operations triggered by the string. It is shown that functions like the replication or transcription of DNA can be acccomplished using such a sequence. Several advantages of the revised design of SeMar are also discussed.

1 Introduction

A biological system is a creative system. Although the organic living things on this planet use enormous amounts of resources (time and space), they have succeeded in adapting to their environments and have evolved various forms of life. This has been a strong motivation in that, until recently, a number of life-like computational systems have been proposed and studied. Genetic algorithms [1, 2, 3, 4], DNA computing [5, 6], chemical computation [7, 8, 9, 10, 11], membrane computing [12, 13, 14], and a number of artificial life systems [15] have been classified into this category, and they have proved that the mechanisms they borrowed from biological systems are in fact efficient and useful for the purpose of computation.

Among them, the author focuses on chemical computation and several core memory-type artificial-life systems, which are relevant to this paper. Chemical computation [7, 8, 9, 10, 11] is an approach of trying to devise a computer system using a comparison between information objects and chemical molecules. A set of objects (denoted by characters or strings) are prepared in a computer memory, and a set of operations are defined between them in the imitation of chemical reactions. Specifically, in the P-system proposed by Paun [12, 13, 14], all objects are put into membrane bound compartments, and computation proceeds with not only operations between objects but also the creation or extinction of membranes. One of the most distinctive features of chemical computation is a *flat* data structure between objects. The character/string set is represented by a mathematical set or multi-set, and these characters/strings are operated in parallel or in a random order. This feature is considered to be a direct representation of chemical reaction processes that proceed essentially in parallel in a solvent.

When we peep into biological reaction processes in a living cell, however, we find that most of the processes in the cell or compartment are executed *locally* using particular molecules densely distributred in that space [16]. Molecules produced by a kind of chemical reaction slowly scatter by diffusion processes in a cell. The diffusion constant is not an infinite number; hence, during this diffusion, most catalytic reactions can be accomplished locally. Another more important strategy taken by a cell involves microtubules. Millions, or even billions, of microtubules do not only constitute the skeleton of a cell, but also help dynamic molecular transfer to proceed. Various kinds of molecules are conveyed along the microtubules stretched around a cell, and without this transfer, the cell cannot even divide itself in a proper manner. These observations suggest that some geographical information attached to a computational object is essential for highly functional operations among objects. We might as well consider a biological cell to be a set of geographically mapped molecules besides considering it to be a flat solvent in which chemicals are dissolved homogeneously.

A core memory [17, 18, 19, 20, 21, 22, 23] is a one-dimensional information space that holds data words (objects) with (geographical) addresses. The addresses align the words in the order of address values, and at the same time, enable a word to access or interact with another particular word. The *apparent* dimension of the core memory is evidently one; however, the *practical* dimension of this space can be measured by the number of nearest neighbors to a word, or in other words, the number of other words that can be directly accessed by one word without translating address numbers. For example, if a core memory is manipulated by a central processing unit (CPU) [19, 20], the number of directly accessed words is the number of address registers implemented in the CPU, so that the practical dimension of the core is given by a half of the number of the registers. For another example, in the case of the dynamic core memory SeMar proposed by the author [21, 22, 23], every word in the core is equipped with two address registers pointing to the previous and next neighboring words, so that the core's practical dimension is one.

As can be easily recognized, however, the dimension of a computational medium is of vital importance for the richness of the functions possibly emerging in the computational system. The dimension (number of directly accessed words) determines the number of operand objects (substrate molecules) operated by an operator object (protein molecules) at once, and more importantly, the dimension determines the number of operator objects that can cooperate simultaneously. In a biological cell, several complicated and important functions such as transcription, splicing, and translation are accomplished by the cooperation of dozens of protein molecules. We cannot expect the emergence of higher functions in a computational system when only a small number of operator objects can cooperate for a single task.

Inspired by this argument, here the author extends the SeMar core to a high-dimensional medium. In addition to two address registers attached to a word, three extra registers pointing to other words are prepared for each word explicitly. The entire core is made up of a number of sections hierarchically structured by pointers, and a section holds a set of data words. Each word is represented by a variable-length string, and the type of string is judged to be under DNA, Protein, Membrane, or others, depending upon the first character. Specifically, a Membrane word can hold a pointer to a lower section, and a Protein word is executed using a pointer that points to an (amino acid) character in the string.

In the following, the author first describes the proposed data structure of a section or a word, together with the conceptual background (Sect. 2). Next, the revised design of SeMar is briefly explained in Sect. 3 focusing on machine operations triggered by characters and the design of a sample creature. The implementation of the total system for an evolutionary experiment has not yet been prepared. Sect. 4 is dedicated to a summary of the proposal and a discussion on its meanings and merits.

2 A Dynamic Core that Represents Tight Molecular Interaction

In order to imitate a biological system in which functionally relevant molecules agglomerate and cooperate, we have to implement tight relations between information objects. In a computational medium, the distance between a pair of objects cannot be measured by the Euclid distance but by the time needed for one object to access the other [24]. When one object can interact with the other object by using a direct value stored in an address register, the pair is mapped close to each other; on the other hand, when one object cannot interact with the other without a number of translations of address numbers, the pair is mapped some distance away from each other. An information object can interact with more objects at once with more address registers. Accordingly, in order to make objects highly cooperate, we have to increase the number of address registers directly manipulated by an object.

Figure 1 shows a case of interaction between words in the SeMar core. To make the words interact like in Fig. 1(b), a word data structure as shown in Fig. 2 is prepared in the revised model. Every word is equipped with five word-address registers named PRV, NXT, OPR, RF1, and RF2, and three character-address registers named E, X, and Y. E always points to a character in the current word, and X and Y always point to characters in a word pointed to by OPR. Although we still think that all words in a section are one-dimensionally aligned in the order of address numbers, the practical dimension of the core is $5/2 = 2.5$ in this implementation.

Furthermore, here we have to note that the address registers shown in Fig. 2 must be prepared explicitly and made 'visible' or 'readable' to other words for two reasons. The first reason is for cooperation between operator words. If, for example, the numbers in the address registers were hidden from other words, operator words would be unable to copy numbers stored in the address registers of other operators, and this would lead operators to cease to cooperate with other operators. The visibilily enables different operator words to have operand words in common and to work for the same task of computation.

The second reason for the visiblily of registers is a more practical one. Although a number of operator words in a SeMar core are executed logically in parallel, they are actually put into action one by one in a simulation run using a standard single-CPU computer. If all current statuses of an operator word are stored in the explicitly

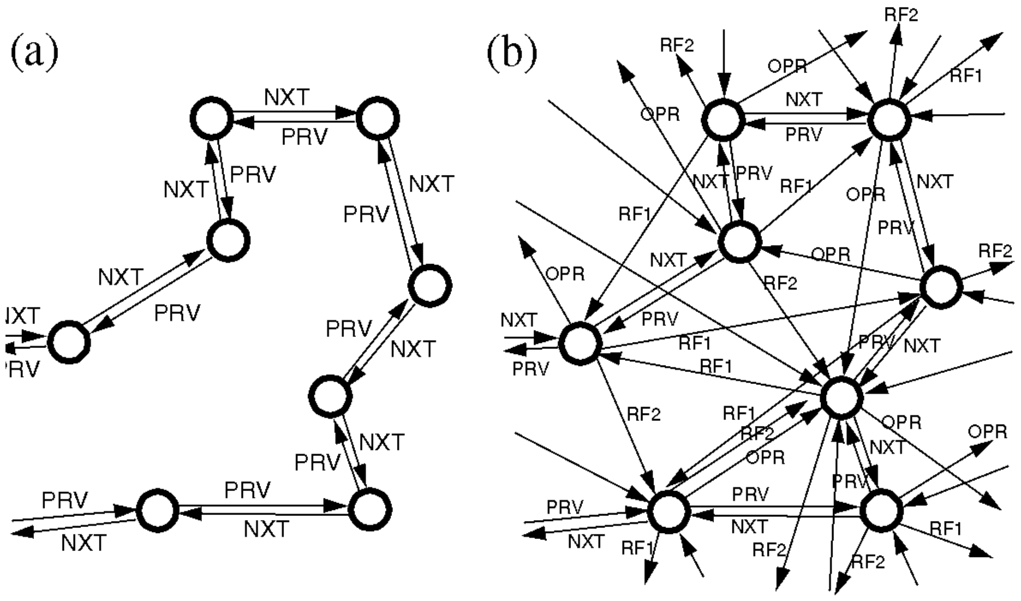


Figure 1: Schematic illustrations of the relation between words in the SeMar core. (a) Former model, and (b) the proposed model. In both figures, a node represents a word (object), and a directed edge represents an access by an address register named PRV, NXT, or so on. (a) A word can interact with two words so that the practical dimension is $2/2 = 1$, and (b) a word can interact with five words so that the practical dimension is $5/2 = 2.5$.

prepared address registers, the CPU can resume its execution without any loss of information after an interruption during which other operator words are executed.

3 The Revised Model of SeMar

The features of the revised model of Semar are summarized as follows.

- The entire core is made up of a number of sections hierarchically structured by pointers at Membrane words. Almost all computational operations are confined within a section.
- A section comprises a set of data words. Each word holds a variable-length string (MMO), a set of address registers (PRV, etc.), and a pointer to a lower section (LSc) (Fig. 2).

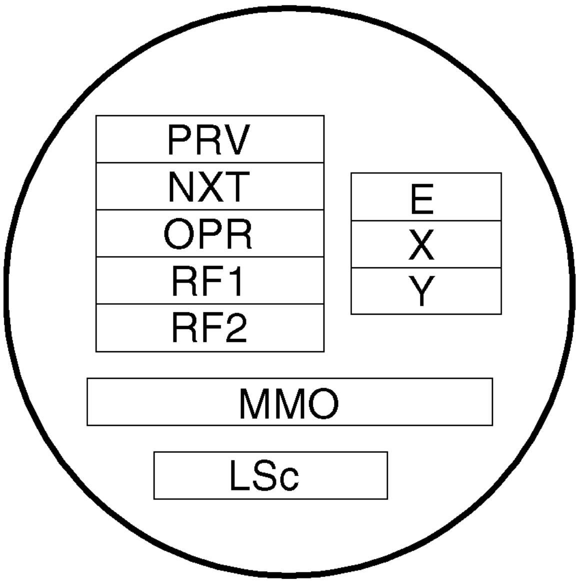


Figure 2: Data structure for one word in the SeMar core. MMO is a variable length string that represents a macromolecule. PRV is an address register pointing to the previous word, NXT is an address register pointing to the next word, OPR is an address register pointing to the operand word, RF1 is an address register pointing to the 1st reference word, RF2 is an address register pointing to the 2nd reference word, E is an address register pointing to an execution cahracter in the present word, and X and Y are address registers pointing to operand characters in the operand word. LSc is a pointer to a lower section which is valid only when the word represents Membrane.

- Depending upon the first character of MMO, all words are classifed into one of four groups: DNA, Protein, Membrane, or others.
- All actions in the core are activated by Protein words that act as operators. The characters in a Protein word are put into action one by one using a character pointer. The proteins' actions are logically parallel.

In the revised implementaion, the SeMar core is not a single consecutive memory but a set of separated sections, each of which has a data structure like that shown in Fig. 1(b).

When the first character of MMO is the starter signal of a Membrane word, LSc has a meaning and can point to a lower section. The creation, fusion, separation, deletion, or movement of sections is also accomplished by particular section-operation characters in Protein words (although they are not fully implemented in the present version).

The actions of the core proceed with a parallel execution of Protein words. When the first character of MMO is a starter signal of a Protein word, a sequence of characters in MMO is put into action by a character pointer E. After each character execution, E is increased by one by default, so that the execution of the Protein string is sequential.

3.1 Elementary Character Functions

Table ?? shows an example design of elementary characters. A string in MMO consists of these characters for any kind of word. The prepared 45 characters are classified into six classes.

Class 0 characters are not put into action. They work as starters of Membrane/DNA/Protein words (for `M`, `D`, or `P`), or work as units constituting a pattern used for the detection of other words or other character regions. Namely, an address-changing character followed by Nop characters uses a consecutive Nop sequence for a pattern and searches for a matched word/region using complementary matching. This pattern matching strategy for determining address registers was first invented by Ray [19] in the imitation of the conformation matching between biological molecules.

Class 1 characters change the word address registers. A character of this class substitutes a word register with the value of another word address register or the address of a word detected by a Nop pattern matching. Note that characters able to change a word address register numerically are not included in this class because such kinds of characters cause a word address register to have a meaningless value. Character `h` is for the cooperation between Proteins. This character makes a Protein copy the values of OPR, RF1, X, and Y from the word at RF2 and has operand data in common with the word at RF2.

Class 3 characters change the values of the character address registers E, X, or Y. In contrast to Class 1 characters, the numerical modifications (reset, increase, or decrease) of character registers are allowed by `n`, `o`, `q`, `r`, etc. Since these registers always point to characters in a Protein/operand word, there is no possibility of their pointing to meaningless characters even if they are modified numerically.

Class 2 and Class 4 characters are in charge of the actual modification of data words in a section. When these characters are put into action, Proteins bring about the creation or transfer of a word, or the insertion, deletion, or modification of a character.

The character in Class 5 is an operation for section modification. In this interim design, there is only one character prepared for this class, but the author plans to prepare several other characters for this class in order to make a SeMar creature able to ingest nutrition from the environment and divide itself.

3.2 An Example Design of a Creature

Using the elementary characters presented in the previous subsection, the author tentatively designed a creature (a section), which is unable to reproduce itself but has several important characteristics in common with a self-reproducing creature to be designed in future. This self-reproducing creature, in its final form, will be used as an ancestor innoculated in the SeMar core at the beginning of an evolutionary run [23]. Here, the author summarizes the characteristics of the designed creature.

Class description	Char. mnem.	Function of a character
Class 0: No operation	`0`	Nop0. Complementarily matches with Nop1.
	`1`	Nop1. Complementarily matches with Nop0.
	`2`	Nop2. Complementarily matches with NopM.
	`3`	Nop3. Complementarily matches with NopD.
	`4`	Nop4. Complementarily matches with NopP.
	`M`	NopM. Complementarily matches with Nop2. Starter of a Membrane word.
	`D`	NopD. Complementarily matches with Nop3. Starter of a DNA word.
	`P`	NopP. Complementarily matches with Nop4. Starter of a Protein word.
Class 1: Oper. for word addresses	`a`	Jumps Protein to after the nearest matched word.
	`H`	OPR $\leftarrow$ (address of the tail of the section)
	`e`	OPR $\leftarrow$ RF1
	`f`	RF1 $\leftarrow$ OPR
	`g`	OPR $\leftarrow$ RF2
	`p`	RF2 $\leftarrow$ OPR
	`h`	OPR/RF1/X/Y $\leftarrow$ OPR/RF1/X/Y of word(RF2)
	`j`	Changes OPR to the add. of a matched word before OPR.
	`k`	Changes RF1 to the add. of a matched word before RF1.
	`l`	Changes RF2 to the add. of a matched word before RF2.
Class 2: Oper. for words	`I`	Moves word(OPR) before RF1.
	`m`	Creates a word with the matched pattern before RF1.

Table 1: Mnemonics of elementary characters and their functions. Here, for example, word(RF1) represents the word at RF1, char(X) represents the character at X in the word at OPR, and pat(X-) represents a consecutive Nop pattern that starts at X in the word at OPR.

Class description	Char. mnem.	Function of a character
Class 3: Oper. for char. addresses	b	Jumps E to a matched pattern.
	d	Jumps E to a matched pattern if X=0.
	c	Jumps E to a matched pattern if X=Y.
	i	Jumps E to a matched pattern if pat(X-)=pat(Y-).
	J	Skips E if a matched word exists in the section.
	n	$X \leftarrow 0$
	o	$X \leftarrow$ (address of the last char. of word(OPR))
	q	$Y \leftarrow X$
	r	$X \leftarrow X - 1$
	s	$X \leftarrow X + 1$
	t	$Y \leftarrow Y - 1$
	u	$Y \leftarrow Y + 1$
	v	Changes X to the add. of a matched pattern before X.
	w	Changes Y to the add. of a matched pattern before Y.
Class 4: Oper. for chars.	x	char(X) $\leftarrow 0$
	y	Inverts the rightmost bit of char(X).
	z	Shifts char(X) left.
	A	Shifts char(X) right.
	B	char(X) $\leftarrow$ char(Y)
	C	char(Y) $\leftarrow$ char(X)
	E	Deletes char(X).
	F	Inserts a copy of char(X) before char(Y).
	G	Inserts char(X) at the head of word(RF1).
	L	Appends char(X) at the tail of word(RF1).
Class 5: Oper. for sections	K	Separates the current section at the border before OPR.

Table 1: (continued)

The detailed sequence of characters is shown in Appendix A using a C source code form, and a sequence of snapshots of the creature's actions is shown in Fig. 3.

(1) The section includes initial two Protein words and one DNA word in which two genes are coded (Fig. 3(a)).

(2) The two initial Protein words are a regulatory protein and a transcription protein. The regulatory protein first changes the transcription capability of the DNA word into 'active', and after that, the transcription protein transcribes the DNA to create Protein words. (Figure 3(b) shows the final snapshot of this procedure.)

(3) The two genes in the DNA word are a gene for DNA replication and a gene for section division. The protein words created from these genes replicate the DNA word, and then, divide the section. (Figure 3(c) shows the final snapshot of this procedure.)

(4) The transcription capability of a DNA word is represented by the second character of a string stored in `MM0`. A regulatory protein that activates a DNA word changes a word beginning with '`D0`' into a word beginning with '`D1`' in a form accessible by the transcription protein.

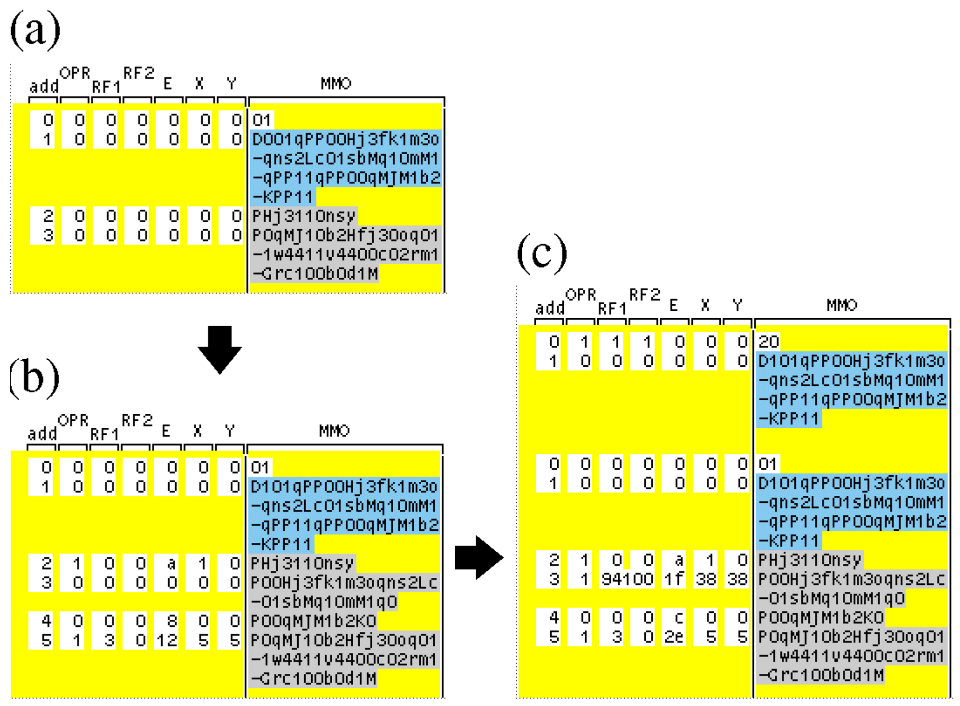
(5) A gene's starting and stopping signals in a DNA word are patterns '`PP00`' and '`PP11`', respectively. The transcription protein creates a new word and copies a substring delimited by these signals to the new word. The first character of the created word is automatically set to '`P`' on account of the copy of the starting signal '`PP00`'.

(6) Operation sequence control between Proteins is achieved by a signal word created by the former Protein. At the initial substring of a Protein that must be inhibited until the execution of another Protein, a small infinite loop by '`b`' is written. The execution address `E` can jump out of this loop by '`J`' when some signal word is created by the former Protein.

4 Discussion

The basic design of the core-memory-type artificial life system SeMar was revised to enhance the interaction between words. A section that is a small compartment of the core is represented by a set of words with address numbers, and each word holds a number of address registers. The register set attached to a word can describe the high-dimensional interaction between words aligned in the one-dimensional core. In the following, several meanings of the revised design are discussed.

High-dimensional interaction in a one-dimensional core. The ability of a multi-agent system like SeMar is largely determined by the richness of interaction between agents (Proteins). In a standard sequential program for a von Neumann



(a)

add	OPR	RF1	RF2	E	X	Y	MMO
0	0	0	0	0	0	0	01
1	0	0	0	0	0	0	D001qPPOOHj3fk1m3o
							-qns2LcO1sbMq1OmM1
							-qPP11qPPOOqMJM1b2
							-KPP11
2	0	0	0	0	0	0	PHj311Onsy
3	0	0	0	0	0	0	POqMJ1Ob2Hfj3Ooq01
							-1w4411v4400c02rm1
							-Grc100b0d1M

(b)

add	OPR	RF1	RF2	E	X	Y	MMO
0	0	0	0	0	0	0	01
1	0	0	0	0	0	0	D101qPPOOHj3fk1m3o
							-qns2LcO1sbMq1OmM1
							-qPP11qPPOOqMJM1b2
							-KPP11
2	1	0	0	a	1	0	PHj311Onsy
3	0	0	0	0	0	0	POOHj3fk1m3oqns2Lc
							-O1sbMq1OmM1qO
4	0	0	0	8	0	0	POOqMJM1b2KO
5	1	3	0	12	5	5	POqMJ1Ob2Hfj3Ooq01
							-1w4411v4400c02rm1
							-Grc100b0d1M

(c)

add	OPR	RF1	RF2	E	X	Y	MMO
0	1	1	1	0	0	0	20
1	0	0	0	0	0	0	D101qPPOOHj3fk1m3o
							-qns2LcO1sbMq1OmM1
							-qPP11qPPOOqMJM1b2
							-KPP11
0	0	0	0	0	0	0	01
1	0	0	0	0	0	0	D101qPPOOHj3fk1m3o
							-qns2LcO1sbMq1OmM1
							-qPP11qPPOOqMJM1b2
							-KPP11
2	1	0	0	a	1	0	PHj311Onsy
3	1	941	00	1f	38	38	POOHj3fk1m3oqns2Lc
							-O1sbMq1OmM1qO
4	0	0	0	c	0	0	POOqMJM1b2KO
5	1	3	0	2e	5	5	POqMJ1Ob2Hfj3Ooq01
							-1w4411v4400c02rm1
							-Grc100b0d1M

Figure 3: A sequence of snapshots of a designed creature. (a) An initial snapshot, (b) a snapshot after completing the transcription of two genes, and (c) a snapshot after replicating a DNA word and performing section division. The contents of the address registers are denoted by decimal integers, and strings in MMO are represented by character mnemonics. A long string is shown with several separated lines using a specific character '-' which represents the continuity of the string.

computer, an agent can be considered to be a subroutine (or a library function), and the richness of the interaction between the subroutines is measured by the number of variables given to or returned from a subroutine. At the level of an assembler program, this number is nearly equal to the number of registers prepared in the employed hardware so that the richness of the interaction between the assembler routines is ensured by a number of registers in the CPU. From this point of view, the former design of SeMar [22, 23] had a serious drawback. A Protein word in the core had only two address registers visible to other Proteins; hence, the practical dimension of the core was the same as the apparent dimension of the core (namely, one). The word design proposed in Fig. 2 has remedied this problem and introduced close cooperation between words in a dynamic core.

Parallel execution of sequentially executed strings. One of the most advantageous features of SeMar is the parallel execution of operator words. A program (DNA words) and operators (Protein words) are prepared apart in the core, and the total execution of the program is accomplished by a number of operators working in parallel. In the presented implementation of SeMar, a Protein word is represented by a sequence of characters executed sequentially; and yet the system cannot become stuck by a problem like an 'evolutionary dead end' [22]. In the SeMar core, a small infinite loop causing an evolutionary dead end always occurs in a single Protein so that the creature's total execution rarely stops if several Proteins become non-functional on account of the dead end.

Estimation of the density of functional protein genotypes. Because evolution can be explored only through a connected network of viable genotypes in a genotype space, the evolvability (defined as the possibility of evolving a variety of genotypes) of an evolutionary system (or a protein) can be measured by the density of viable (or functional) genotypes in the genotype space [25, 26, 27]. When we consider the evolvability of an artificial protein on this notion, we first have to clarify the definition of the 'functionality' of a digital protein. In the revised design of SeMar, one appropriate definition of a functional protein is evidently a string wherein the character pointer (E) does not become stuck by a small infinite loop and can trace a fairly large portion of the string. By determining a particular threshold trace-rate value above which a protein is judged to be functional and by estimating the density of functional proteins with some numerical method, we can assess the design of elementary characters quantitatively. Optimizing the design of SeMar using this assessment is a future study to be tackled.

Acknowledgements

The author would like to thank Dr. K. Shimohara of ATR-International for his continual encouragement.

Appendix A: Character Codes of a Designed Creature

```
{{ //################### NUT #####################//
'0',   // Nop0
'1',   // Nop1
'\0'},

{ //################### DNA #####################//
'D',   // NopD
'0',   // Nop0                                  ('1': transcibable)
'0',   // Nop0                                     (operon signal)
'1',   // Nop1                                     (operon signal)
'q',   // Y ← X                                           (dummy)

//////////////////////////// Gene for Replicating DNA ////////////////////////////
'P',   // NopP
'P',   // NopP
'0',   // Nop0
```

```
'0',  // Nop0

'H',  // OPR ← tail
'j',  // jump OPR                                          (OPR = add of DNA)
'3',  // Nop3
'f',  // RF1 ← OPR
'k',  // jump RF1                                       (RF1 = add of word [0-])
'1',  // Nop1
'm',  // create word before RF1                        (create a word for DNA)
'3',  // Nop3                                        (RF1 = add of created DNA)
'o',  // X ← (add of tail)
'q',  // Y ← X                                             (Y = (add. of tail))
'n',  // X ← 0
's',  // X ++                                                         (X = 1)

'2',  // Nop2
'L',  // append char(X) at the tail of RF1                         (copy DNA)
'c',  // jump E if X=Y
'0',  // Nop0
'1',  // Nop1
's',  // X ++
'b',  // jump E                                                  (go to Nop2)
'M',  // NopM

'q',  // Y ← X                                                      (dummy)
'1',  // Nop1
'0',  // Nop0
'm',  // create word before RF1          (create terminating signal word [20])
'M',  // NopM
'1',  // Nop1
'q',  // Y ← X                                                      (dummy)

'P',  // NopP
'P',  // NopP
'1',  // Nop1
'1',  // Nop1

'q',  // Y ← X                                                      (dummy)

//////////////////////////// Gene for Splitting Section ////////////////////////////
'P',  // NopP
'P',  // NopP
'0',  // Nop0
'0',  // Nop0

'q',  // Y ← X                                                      (dummy)
'M',  // NopM
'J',  // skip E if pat word exists            (jump out of an infinite routine)
'M',  // NopM                                          (activated by word [20]
'1',  // Nop1
'b',  // jump E                                                (go to pat [M])
```

```
'2',   // Nop2                                        (make an infinite routine)

'K',   // sep SCTN before OPR

'P',   // NopP
'P',   // NopP
'1',   // Nop1
'1',   // Nop1

'\0'},

{ //################### PRO #####################//
// Regulatory Protein : changes [D001] into [D101]
'P',   // NopP
'H',   // OPR ← tail
'j',   // jump OPR                                    (OPR = add of word [D001])
'3',   // Nop3
'1',   // Nop1
'1',   // Nop1
'0',   // Nop0

'n',   // X ← 0
's',   // X ++                                                          (X = 1)
'y',   // char(X) ← char(X)∧00000001
'\0'},

{ //################### PRO #####################//
// Transcription Protein : transcribes DNA to create Protein
'P',   // NopP
'0',   // Nop0
'q',   // Y ← X                                                        (dummy)
'M',   // NopM
'J',   // skip E if pat word exists             (jump out of an infinite routine)
'1',   // Nop1                                          (activated by word [01]
'0',   // Nop0
'b',   // jump E                                                 (go to pat [M])
'2',   // Nop2                                        (make an infinite routine)

'H',   // OPR ← tail
'f',   // RF1 ← OPR                                           (RF1 = add of tail)
'j',   // jump OPR                                      (OPR = add of word [D1])
'3',   // Nop3
'0',   // Nop0
'o',   // X ← (add of tail)
'q',   // Y ← X                                           (X = Y = add of tail)

'0',   // Nop0                                    (transcribe a Protein region)
'1',   // Nop1
'1',   // Nop1
'w',   // jump Y prev                                   (Y = add of pat [PP00])
'4',   // Nop4
'4',  // Nop4
'1',  // Nop1
'1',  // Nop1
'v',  // jump X prev                               (X = add of pat [PP11])
'4',  // Nop4
'4',  // Nop4
'0',  // Nop0
'0',  // Nop0
'c',  // jump E if X=Y                                       (go to ending)
'0',  // Nop0
'2',  // Nop2
'r',  // X --

'm',  // create word before RF1                   (create a word for PRO)
'1',  // Nop1
'G',  // ins char(X) at the head of RF1                    (copying DNA)
'r',  // X --
'c',  // jump E if X=Y                                (go to next Protein)
'1',  // Nop1
'0',  // Nop0
'0',  // Nop0
'b',  // jump E
'0',  // Nop0

'd',  // jump E if X=0                                           (dummy)
'1',  // Nop1                                                   (ending)
'M',  // NopM
'\0'}};
```

References

[1] Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT Press, Boston (1992)

[2] Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, New York (1989)

[3] Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Boston (1996)

[4] Suzuki, H., Iwasa, Y.: Crossover Accelerates Evolution in GAs with a Babel-like Fitness Landscape: Mathematical Analyses. Evolutionary Computation **7**(3) (1999) 275-310

[5] Adleman, L.: Molecular Computation of Solutions to Combinatorial Problems. Science, **266** (1994) 1021-1024

[6] Paun, G., Rozenberg, G., Salomaa, A.: DNA Computing. New Computing Paradigms, Springer-Verlag, Heidelberg (1998)

[7] Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science **96** (1992) 217–248

[8] Fontana, W.: Algorithmic chemistry. In: Langton, C.G. et al. (eds.): Artificial Life II: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems (Santa Fe Institute Studies in the Sciences of Complexity, Vol. 10). Addison-Wesley (1992) 159–209

[9] Banzhaf, W.: Self-organization in a system of binary strings. In: Brooks, R., Maes, P. (eds.): Artificial Life IV: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems. MIT Press, Cambridge (1994) 109–118

[10] Dittrich, P., Banzhaf, W.: Self-evolution in a constructive binary string system. Artificial Life **4** (1998) 203–220

[11] Suzuki, Y., Tanaka, H.: Chemical evolution among artificial proto-cells. To be published in Artificial Life VII Proceedings (2000)

[12] Paun, G.: Computing with membranes. Turku Centre for Computer Science Technical Report No. 208 ISBN 952-12-0303-X (1998)

[13] Paun, G.: Computing with membranes (P-systems): Twenty six research topics. Auckland University, CDMTCS Report No 119 (2000) Available at http://www.cs.auckland.ac.nz/CDMTCS

[14] Castellanos, J., Paun, G., Rodriguez-Paton, A.: Computing with membranes: P-systems with worm-objects. Submitted (2000)

[15] Adami, C.: Introduction to Artificial Life. Springer-Verlag, Santa Clara, CA (1998)

[16] Alberts, B., Bray, D., Lewis, J., Raff, M., Roberts, K., Watson, J.D.: Molecular Biology of the Cell, The third Edition. Garland Publishing, New York (1994)

[17] Rasmussen, S., Knudsen, C., Feldberg, R., Hindsholm, M.: The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. Physica D **42** (1990) 111–194

[18] Rasmussen, S., Knudsen, C., Feldberg, R.: Dynamics of programmable matter. In: Langton, C.G. et al. (eds.): Artificial Life II: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems (Santa Fe Institute Studies in the Sciences of Complexity, Vol. 10). Addison-Wesley (1992) 211–254

[19] Ray, T.S.: An approach to the synthesis of life. In: Langton, C.G. et al. (eds.): Artificial Life II: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems (Santa Fe Institute Studies in the Sciences of Complexity, Vol. 10). Addison-Wesley (1992) 371–408

[20] Ray, T.S.: Selecting Naturally for Differentiation. In: Koza, J.R. et al. (eds.): Genetic Programming 1997: Proceedings of the Second Annual Conference. Morgan Kaufmann, San Francisco (1997) 414–419

[21] Suzuki, H.: One-dimensional unicellular creatures evolved with genetic algorithms. In: JCIS '98: The Fourth Joint Conference on Information Sciences, Proceedings Vol. II. Association for Intelligent Machinery Inc., USA (1998) 411–414

[22] Suzuki, H.: An Approach to Biological Computation: Unicellular Core-Memory Creatures Evolved Using Genetic Algorithms. Artificial Life **5** N.4 (2000) 367–386

[23] Suzuki, H.: Evolution of Self-reproducing Programs in a Core Propeled by Parallel Protein Execution. To be published in Artificial Life **6** N.2 (2000) 103–108

[24] Ray, T.S.: An evolutionary approach to synthetic biology: Zen and the art of creating life. Artificial Life **1** (1994) 179–209

[25] Suzuki, H.: Minimum Density of Functional Proteins to Make a System Evolvable. In: Sugisaka, M., Tanaka, H. (eds.): Proceedings of The Fifth International Symposium on Artificial Life and Robotics (AROB 5th '00) Vol. 1 (2000) 30-33

[26] Suzuki, H.: Evolvability Analysis Using Random Graph Theory. Proceedings of AFSS 2000 (The Fourth Asian Fuzzy Systems Symposium) Vol. 1 (2000) 549-554

[27] Suzuki, H.: Evolvability Analysis: Distribution of Hyperblobs in a Variable-Length Protein Genotype Space. To be published in Artificial Life VII Proceedings (2000)

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 265 - 285.

Artificial Life and P Systems

Yasuhiro Suzuki and Hiroshi Tanaka
Bio-Informatics,
Medical Research Institute,
Tokyo Medical and Dental University
Yushima 1-5-45, Bunkyo, Tokyo 113 JAPAN

Abstract

Artificial chemical system is well studied in Artificial Life and Complexity. Here we introduce a new atificial chemical system: Abstract Rewriting system on Multisets (ARMS). This system belongs to P Systems and is able to show complex phenomena such as non-linear oscillations. We introduce a membrane that is composed of "chemical compounds (denoted by symbols)", and the compounds are generated through chemical reactions in the cell. Furthermore, we apply a genetic method to the system and find some interesting results.

1 Introduction

Artificial Life is the study of man-made systems that exhibit behaviors characteristic of natural living systems. It complements the traditional biological sciences concerned with the *analysis* of living organisms by attempting to *synthesize* life-like behaviors within computers and other artificial media [8]
– C.Langton, *Artificial Life.*

It has been over a decade since Langton proposed "Artificial Life (ALife)" with the above definition. He also explained "by extending the empirical foundation upon which biology is based *beyond* the carbon-chain life that has evolved on Earth, Artificial Life can contribute to theoretical biology by locating *life-as-we-know-it* within the larger picture of *life-as-it-could-be*"

With the advent in biotechology and increasing interests in genome information worldwide, genome databases progress so fast that the whole sequence database of human genome will be published soon. Hence now we are confronted with the problem of how to deal with the overwhelming genome sequences. As traditional biological analysis is only powerful in analysing a single gene most of the case, ALife study turns out to be more and more important in carbon-chain life science.

One challenging and important theme in genome science is to discover gene-networks. Our knowledge of the genes' regulation has come from an accumulation of funamental studies over the past. However recently with a new technology, the DNA chip provides us with genome wide information fast and informative. Combination with genomic database and use of ALife, now we are closer to find the correlation and regulation between genes and genes. To clarify gene network, we usually need to model a system by synthesizing the empirical data at first, in which Alife approach is a necessary. One of an ALife model,Kauffuman's boolean network model,has been a instructive model in analysis gene network[16].

However the subjects of ALife studies are changing. It is much easier for us to obtain various and lots of data from living systems these days, though we are still want of method to observe or analyse systematically. Therefore it is necessary for us to construct a system meanwhile. This resemble the studies in cosmology. Although we can not observe Big-bang in fact, we can infer many features from empirical data.

The way of ALife study is needed in carbon-chain life science in these days. Whole human genome will be analyzed very soon and many other projects of analyzing whole genome are performed. Thus, we will obtain huge genome data soon. But, in these projects we do not analyze genome in fact but just read them. Thus, we can not know how it works as a whole even if the all projects are completed.

Hence, the most challenging and important theme in genome science is to discover genome networks. In order to find it, a knockout mouse has been used and it is not easy to know the interactions between genes. But in these days very powerful technique, the DNA chip is invented[4] and we can easily see the correlation between genes and genes. In order to construct a gene network, we model a system by synthesizing the empirical data. Thus, the way of Alife study is needed. In fact, Kauffuman's boolean network model[7], one of an ALife model, is used to model a gene network[16].

Now ALife studies are changing. We can obtain various and huge data of living systems easily but they are broken pieces, thus we have to contract a system upon them. In the way of study, it looks like Cosmology. Although we can not observe Big-bang in fact, from empirical data its existence is conjectured.

Emperically, abstract Chemical system turns to be powerful in describing a complex system. In this contribution, we first introduce the abstract chemical model we use that bases on a multiset rewriting, presented with some experimental results. We will also give a brief introduction of its application in an ecological system. Then we show the application of the system by introducing membrane structure, which is closely related to P Systems.

Abstract Chemical System

Life can be considered as a system in a specific class of chemical reaction systems, but real biochemical systems are so complex that it is difficult to reconstruct the precise dynamics in the system. Thus, it is important to abstract the essential properties of biochemical systems in order to obtain insights into its dynamical properties.

There have been developed various Artificial Chemistries. In broad sence, an artificial chemistry is a man-made system which similar to a chemical system. It can be defined as a set of objects and a set of reaction rules which specify how the objects interact[?]. It covers large field;

- **Modeling:** living systems (e.g. origin of life), sociology and parallel processes,
- **Information processing:** control, automatic proving, chemical computing,
- **Optimization:** Combinatorial problems (e.g. TSP).

2 *ARMS*

We develop an abstract computational model (ARMS), which can deal with systems with many degrees of freedom and confirm that it can simulate the emergence of complex cycles such as chemical oscillations that are often found in the emergence of life. We also study mathematical properties of the model by using a computational algebra and propose an order parameter to describe the global behavior of the system.

We will introduce the multiset rewriting system, "*Abstract Rewriting system on MultiSets*" in this section. Intuitively, *ARMS* is like a chemical solution in which *molecules* floating on it can interact with each other according to reaction rules. Technically, a chemical solution is a finite multi-set of elements denoted by $A^k = \{a, b, \ldots,\}$; these elements correspond to *molecules*. Reaction rules that act on the molecules are specified in *ARMS* by rewriting rules. As to the intuitive meaning of *ARMS*, we refer to the study of chemical abstract machines [3]. In fact, this system can be thought of as an underlying "*algorithmic chemistry* [2]."

Let A be an *alphabet* (a finite set of abstract symbols). The set of all strings over A is denoted by A^*; the empty string is denoted by λ. (Thus, A^* is the free monoid generated by A under the operation of concatenation, with identity λ.) The length of a string $w \in A^*$ is denoted by $|w|$.

A *rewriting rule* over A is a pair of strings (u, v), $u, v \in A^*$. We write such a rule in the form $u \rightarrow v$. Note that u and v can also be empty. A *rewriting system* is a pair (A, R), where A is an alphabet and R is a finite set of rewriting rules over A.

With respect to a rewriting system $\gamma = (A, R)$ we define over A^* a relation $\Longrightarrow$ as follows: $x \Longrightarrow y$ iff $x = x_1 u x_2$ and $y = x_1 v x_2$, for some $x_1, x_2 \in A^*$ and $u \rightarrow v \in R$. The reflexive and transitive closure of this relation is denoted by $\Longrightarrow^*$. A string $x \in A^*$ for which there is no string $y \in A^*$ such that $x \Longrightarrow y$ is said to be an *dead* one (in other words, from a dead string no string can be derived by means of the rewriting rules).

From now on, we work with an alphabet A whose elements are called *objects*; the alphabet itself is called a *set of objects.*

A *multiset* over a set of objects A is a mapping $M : A \longrightarrow \mathbf{N}$, where $\mathbf{N}$ is the set of natural numbers, 0, 1, 2,.... The number $M(a)$, for $a \in A$, is the *multiplicity* of object a in the multiset M. Note that we do not accept here an infinite multiplicity. The set $\{a \in A \mid M(a) > 0\}$ is denoted by *supp*(M) and is called the *support* of M. The number $\sum_{a \in A} M(a)$ is denoted by *weight*(M) and is called the *weight* of M.

We denote by $A^{\#}$ the set of all multisets over A, including the empty multiset, $\emptyset$, defined by $\emptyset(a) = 0$ for all $a \in A$.

A multiset $M : A \longrightarrow \mathbf{N}$, for $A = \{a_1, \ldots, a_n\}$, can be naturally represented by the string $a_1^{M(a_1)} a_2^{M(a_2)} \ldots a_n^{M(a_n)}$ and by any other permutation of this string. Conversely, with any string w over A we can associate a multiset: denote by $|w|_{a_i}$ the number of occurrences of object a_i in w, $1 \leq i \leq n$; then, the multiset associated with w, denoted by M_w, is defined by $M_w(a_i) = |w|_{a_i}, 1 \leq i \leq n$.

The union of two multisets $M_1, M_2 : A \longrightarrow \mathbf{N}$ is the multiset $(M_1 \cup M_2) : A \longrightarrow \mathbf{N}$ defined by $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$, for all $a \in A$. If $M_1(a) \leq M_2(a)$ for all $a \in A$, then we say that multiset M_1 is included in multiset M_2 and we write $M_1 \subseteq M_2$. In such a case, we define the multiset difference $M_1 - M_2$ by $(M_2 - M_1)(a) = M_2(a) - M_1(a)$, for all $a \in A$. (Note that when M_1 is not included in M_2, the difference is not defined).

A rewriting rule such as

$$a \rightarrow a \ldots b,$$

is called a *heating rule* and denoted as $r_{\triangle > 0}$; it is intended to contribute to the stirring solution. It breaks up a complex *molecule* into smaller ones: *ions*. On the other hand, a rule such as

$$a \ldots c \rightarrow b,$$

is called a *cooling rule* and denoted as $r_{\triangle < 0}$; it rebuilds *molecules* from smaller ones. In this paper, reversible reactions, i.e., $S \rightleftharpoons T$, are not considered. We shall not formally introduce the refinement of *ions* and *molecules* though we use refinement informally to help intuition (on both types of rules we refer to [3]).

A *multiset rewriting rule* (we also use to say, *evolution rule*) over a set A of objects is a pair (M_1, M_2), of elements in $A^{\#}$ (which can be represented as a rewriting rule $w_1 \rightarrow w_2$, for two strings $w_1, w_2 \in A^*$ such that $M_{w_1} = M_1$ and $M_{w_2} = M_2$). We use to represent such a rule in the form $M_1 \rightarrow M_2$.

An *abstract rewriting system on multisets* (in short, an *ARMS*) is a pair

$$\Gamma = (A, R)$$

where:

1. A is a set of objects;
2. R is a finite set of multiset evolution rules over A;

With respect to an *ARMS* Γ, we can define over $A^{\#}$ a relation: $(\Longrightarrow)$: for $M, M' \in A^{\#}$ we write $M \Longrightarrow M'$ iff

$$M' = (M - (M_1 \cup \ldots \cup M_k)) \cup (M'_1 \cup \ldots \cup M'_k,)$$

for some $M_i \rightarrow M'_i \in R, 1 \leq i \leq k, k \geq 1$, and there is no rule $M_s \rightarrow M'_s \in R$ such that $M_s \subseteq (M - (M_1 \cup \ldots \cup M_k))$; at most one of the multisets $M_i, 1 \leq i \leq k$, may be empty.

With respect to an *ARMS* $\Gamma = (A, R)$ we can define various types of multisets:

- A multiset $M \in A^{\#}$ is *dead* if there

  is no $M' \in A^{\#}$ such that $M \Longrightarrow M'$ (this is equivalent to the fact that there is no rule $M_1 \rightarrow M_2 \in R$ such that $M_1 \subseteq M$).

- A multiset $M \in A^{\#}$ is *initial* if there is no $M' \in A^{\#}$ such that $M' \Longrightarrow M$.

2.1 How ARMS works

In this example, we assume that a will be inputted on each rewriting step, the *maximal multiset size* is 4 and the initial state is given by $\{a, a, f, a\}$. The set of the rewriting rules, Ru_1 is $\{r_1, r_2, r_3, r_4\}$, where each rule is described by the following:

$$aaa \rightarrow b : r_1,\ b \rightarrow a : r_2,\ b \rightarrow c : r_3,\ a \rightarrow bb : r_4.$$

In this example, we assume that rules are selected as following the order $\{r_4 \Rightarrow r_1 \Rightarrow r_3 \Rightarrow r_2\}$. Then, each rule is applied in the following way. First, r_4 is applied. Next, as steps 2 and 3, r_1 and r_3 are applied, respectively. Finally, as step4, r_2 is applied.

$$\begin{array}{cl}
\{aafa\} & \subseteq a \text{ (the left hand side of } r_4) \\
\downarrow & \text{.... can not input } a \text{ and can not apply } r_4, \\
\{aafa\} & \subseteq aaa \text{ (the left hand side of } r_1) \\
\downarrow & \text{.... can not input } a \text{ but can apply } r_1 \\
\{ba\} &
\end{array}$$

Figure 1: Example of rewriting steps of ARMS

Figure 1 illustrates two rewriting steps of the calculation from the initial state.

In the first step, since the base number of the multiset is 4, the system can not input a. On the left hand side of r_4, a is included in $\{aafa\}$, however, it can not use r_4. If a is replaced with bb, the base number of the multiset becomes 5 and it exceeds the *maximal multiset size*, 4.

In the next step, the system can not input a, however, r_1 can apply to the multiset and $\{aafa\}$ is rewritten into $\{ba\}$. Because if aaa is replaced with b, the base number of the multiset does not exceed the *maximal multiset size* (Figure 1).

In step 3, ARMS inputs a to the multiset and transforms it to $\{c, a, a\}$ with r_3.

$$Step3 : \{c, a, a\}.$$

In step 4, the system inputs a, but r_2 can not apply to it. Thus $\{c, a, a\}$ becomes $\{c, a, a, a\}$.

$$Step4 : \{c, a, a, a\}.$$

Typical Examples In this paragraph, we shall present two examples. Let us assume a set of rewriting rule Ru_1 and a *maximal multiset size* is 4. The first example is a case where ARMS generates two cycles. This example has the following rule order:

$$\{r_4 \Rightarrow r_3 \Rightarrow r_2 \Rightarrow r_4 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow r_1 \Rightarrow r_3 \Rightarrow r_4\},$$

whose state transition is shown in Figure 2. After 8 steps, the system forms two cycles, whose periods are of 3 steps.

0.	$\{f\}$	
1.	$\{a, f\}$	
2.	$\{a, a, f\}$	
3.	$\{a, a, a, f\}$	
4.	$\{b, f\}$	↑
5.	$\{a, b, f\}$	a cycle
8.	$\{a, a, a, f\}$	↓
9.	$\{b, f\}$	↑
10.	$\{a, b, f\}$	a cycle
11.	$\{a, a, a, f\}$	↓
12.	$\{b, f\}$	

Figure 2: Example of a system that generates cycles

The next example is a case where ARMS terminates. Although ARMS applies the same rules, the obtained result is completely different (Figure 3). This example has the following rule order:

$$\{r_4 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow r_4 \Rightarrow r_3 \Rightarrow r_1 \Rightarrow r_2\}.$$

The state transition is shown in Figure 3:

3 Experimental results of the simulation of ARMS

We simulated ARMS with various different setups; in this paper we shall discuss two of them as follows:

- Simple setup
- Brusselator model.

Through these experiments, we confirmed that the system is capable of generating complex patterns.

0.	$\{f\}$
1.	$\{a, f\}$
2.	$\{a, a, f\}$
3.	$\{a, a, a, f\}$
4.	$\{b, f\}$
5.	$\{a, b, f\}$
6.	$\{a, a, b, f\}$
7.	$\{a, a, c, f\}$.

Figure 3: Example of a system that halts

3.1 A simulation with A simple setup

Computational experiments were made under the following initial conditions:

1. only five symbols $\{a, b, c, d, e\}$ were used to describe THE rewriting rules,
2. $\{a\}$ was the only input symbol,
3. the *maximal multiset size* was 10,
4. six rules were used for rewriting steps, and
5. two important parameters, namely the frequency of inputs and randomness of rule application, were given for each simulation, where the former four conditions were fixed and two parameters in the last condition were set to variables.

Although these are clearly very simple settings, the experiments led to the following two interesting results. The first one showed the emergence of a *cycle* even under simple initial conditions, compared with Kauffman's network model [7] or Fontana's λ-calculus [2], which both need large-scale computation to generate cyclic structures from a given system. The second result showed the complex behavior of cycles. Fusion of several cycles and period-doubling were observed easily, when randomness in the input was introduced. Figure 3.1 shows a system undergoing period-doubling. (For more details, the reader should refer to Suzuki and Tanaka [18]).

3.2 The Brusselator model of ARMS

In order to confirm that ARMS works as an abstract chemical system, we performed an experiment implementing the Brusselator model [12] within ARMS. The Brusselator is a well-known mathematical model of chemical oscillations of the Belousov-Zabotinsky reaction [6] (see Figure 5.)

We can view the abstract chemical reaction equations as rewriting rules, as Figure 6 shows:

In this simulation, the reaction rate corresponds to the frequency of rule application. If r_1 has the highest reaction rate, then r_1 is applied at the highest frequency.

Simulation of the Brusselator model Let us examine the relationship between the frequency of rule application (reaction rate) and the concentration X and Y in the multiset. The concentration of X and Y in the multiset is indicated by the number of X and Y present in the multiset.

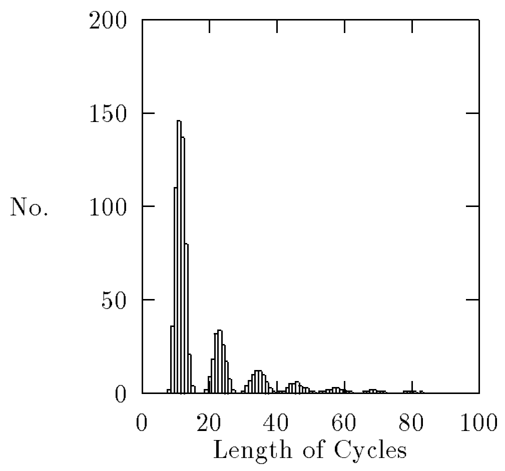


Figure 4: Example of period-doubling

$$
\begin{array}{lllll}
 & \mathrm{A} & \xrightarrow{k_1} & \mathrm{X} & \\
\mathrm{B} & +\mathrm{X} & \xrightarrow{k_2} & \mathrm{Y} & +\mathrm{D} \\
2\mathrm{X} & +\mathrm{Y} & \xrightarrow{k_3} & 3\mathrm{X} & \\
 & \mathrm{X} & \xrightarrow{k_4} & \mathrm{E} & .
\end{array}
$$

Figure 5: Abstract chemical model of the Brusselator.

As to the initial condition, we assume that the *maximal multiset size* is equal to 5000 and the initial state of the multiset is an empty multiset. We assume that the system makes inputs A and B continually. Hence this model can be regarded as a continuously-fed stirred tank reactor (CSTR).

In this simulation, we confirmed that oscillations between the number of X and Y in the multiset emerged. Furthermore, we discovered three types of oscillations as follows: (1) quasi-stable oscillations (Figure 8), (2) unstable oscillations (Figure 9) and (3) divergence and convergence (Figure 7). For further details of this simulation, see [19].

4 Modeling an ecological system by using ARMS

A phenomenon that plants respond to herbivore feeding activities by producing volatiles that in turn attract carnivores enemies of the herbivores has been reported recently[1, 9]. These volatiles are not the mere result of mechanical damage, but are produced by the plant as a specific response to herbivore damage.

In mathematical ecological studies concerning with the system, one notable study is that of, Sabelis and de Jong(1996)[15] who reported that, when herbivore-induced volatiles profitable for plants, the kinds of the volatiles become polymorphic within species. They use game theory and show that ESS corresponds to the case when each species of plant produces the volatiles in polymorphic way.

In order to investigate the population dynamics of the tritrophic systems, we introduce an abstract rewriting system on multisets, Abstract Rewriting System on Multisets (ARMS).

$$
\begin{array}{ccccccccc}
 & & A & \longrightarrow & X & & & : r_1 \\
 & B & X & \longrightarrow & Y & D & & : r_2 \\
X & X & Y & \longrightarrow & X & X & X & : r_3 \\
 & & X & \longrightarrow & E & & & : r_4 .
\end{array}
$$

Figure 6: Rewriting rules for the Brusselator model.

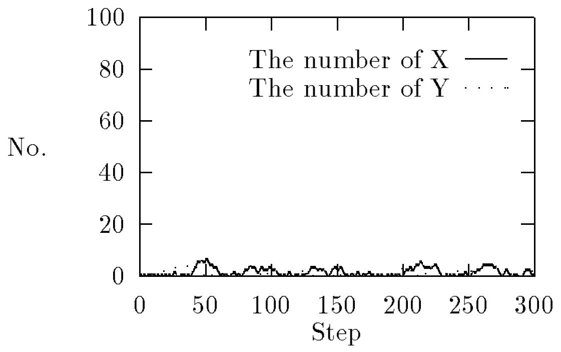


Figure 7: Example of convergence

In ARMS model, we regarded a tritrophic interaction mediated by herbivore-induced plant volatiles that attract carnivorous natural enemies of herbivores as chemical reactions of four reagents (plants, herbivores, carnivores and volatiles). The intensity of interactions between individuals corresponded to reaction speed in the ARMS model. We compared the case where plants produce herbivore-induced volatiles vs. the case where they do not with the model. Further, by changing the reaction speed, we found that there was a case where herbivore-induced volatiles that attract carnivores resulted in the population increase of the herbivores. In ARMS, a reaction rate is realized as the frequency of applying a rule.

Implementing the system by using *ARMS* We assume the symbol "*a*" as a leaf, "*b*" as a herbivore, "*d*" as a carnivorous and "*c*" as a certain density of herbivore-induced volatiles that attract carnivores, respectively. Furthermore, we add "*e*" as an "empty state" in order to introduce "death state." A plant is defined implicitly as the certain number of leaves. Evolution rule R_1 is defined as follows;

$$
\begin{array}{rcll}
a & \xrightarrow{k_1} & aa & r_1 \quad (increasing\ \ leaves), \\
ab & \xrightarrow{k_2} & bbc & r_2 \quad (a\ herbivore\ eats\ a\ leaf), \\
dbc & \xrightarrow{k_3} & dd & r_3 \quad (a\ carnivou\ catches\ herbivore), \\
d & \xrightarrow{k_4} & e & r_4 \quad (the\quad death\ of\ a\ carnivous), \\
b & \xrightarrow{k_5} & e & r_5 \quad (the\quad death\ of\ a\ herbivore).
\end{array}
$$

$k_1 ... k_5$ denotes *reaction rate* that correspond to the frequency of rule application. For example, when $k_4 = 0.1$ and $k_5 = 0.2$ then r_5 will be applied twice as much as r_4. k_2 is

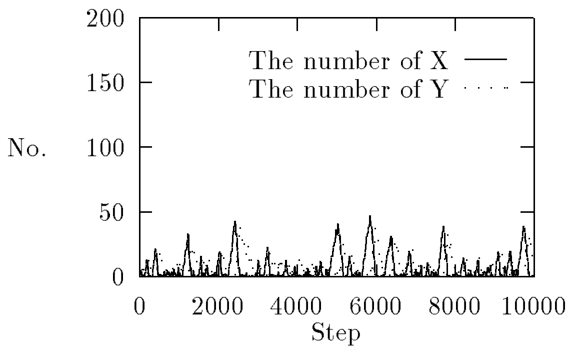


Figure 8: Example of (quasi) stable oscillation

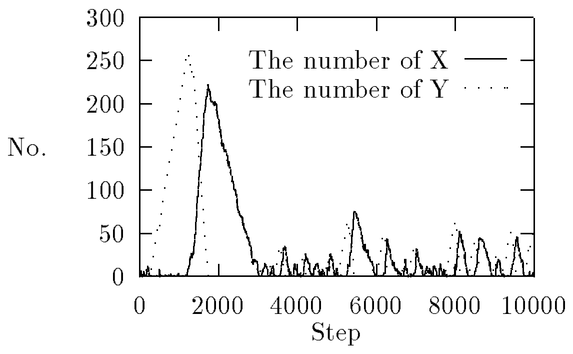


Figure 9: Example of unstable oscillation

defined according to the state of multiset. It is defined as follows;

$$k_2 = \frac{M(b)}{M(a)+M(b)+M(c)+M(d)+M(e)}.$$

The r_1 corresponds to sprout and growth of a plant, r_2 to the case when a herbivore eats a leaf and the leaf generates volatiles, r_3 to a carnivorous catches herbivore, r_4 to the death of a carnivorous and r_5 to the death of a herbivore, respectively. The r_2 denotes the case when there exists a leaf (a), a herbivore eats the leaf and breeds there (b). Then the leaf produces volatiles compound (c) that will attract carnivorous. r_3 denotes the case when there is a herbivore (b) with the volatiles (c), a carnivorous "d" is attracted by it and catches the herbivore and breed there (dd). The breeding rate of carnivorous is expressed as changing the number of right hand side of ds, such as $dbc \xrightarrow{k_2} dddd$.

By using this model, we compared the case when leaves generate volatiles to does not. The evolution rules of the system without volatiles R_2 is defined as follows;

$$\begin{aligned} a &\xrightarrow{k_6} aa \quad k_6 \quad (\textit{increasing leaves}), \\ ab &\xrightarrow{k_7} bb \quad k_7 \quad (\textit{a herbivore eats a leaf}), \end{aligned}$$

$$\begin{aligned} db &\xrightarrow{k_8} dd \quad k_8 \quad (a\ carnivou\ catches\ herbivore), \\ d &\xrightarrow{k_9} e \quad k_9 \quad (the \quad death\ of\ a\ carnivorous), \\ b &\xrightarrow{k_{10}} e \quad k_{10} \quad (the \quad death\ of\ a\ herbivore). \end{aligned}$$

We set reaction rates of k_1, k_4, k_5, k_6, k_9 and k_{10} as 0.5, 0.1, 0.1, 0.5, 0.1 and 0.1, respectively.

Throughout the simulation, we discovered that herbivores could keep their population for at least 1000 generations in the system where the infested leaves generated the volatiles, whereas the herbivores were exterminated by the carnivores in around 100 generations in the system where leaves did not do so. In our model, the carnivores that use the volatiles to find the herbivores are not able to find their victims in a plant from which the volatiles were emitted under the detectable level for carnivores. This is probably the reason why the herbivores survived in the system where leaves generated the carnivore attractant.

This result suggests the possibility that herbivores induce the volatiles for their survival. This may be true in the tritrophic system consisting of plants, two-spotted spider mites (Tetranychus urticae) and predatory mites (Phytoseiulus persimilis) (Dicke et al. 1998). Two-spotted spider mite is a tiny (ca. 0.6 mm) herbivore. However, due to their rapid population increase on a plant, they tend to overexploit the plant. A kidney bean leaf infested by the spider mites started emitting volatiles that attract predatory mites P. persimilis (Maeda et al. 1998). Once in the prey colony, P. persimilis overexploit the spider mites. However, the volatiles were induced only after the spider mites increased over certain number per leaf (ca. more than 100-300 females per small plant) (Maeda et al. in prep) and there will be a time lag between the emission of the volatiles and the visitation by the predatory mites. Thus, the spider mites of the next generation that disperse from the current patch before the plant start emitting the volatiles can be free from the predator. Such spider mites will establish a new colony nearby. At the same time, the original colony may be exterminated by the predators. This cycle may be one of the defense strategies of the two-spotted spider mites against the predatory mites that search for them with the volatiles, and may be applicable for the prediction of the model. We will test this in the future experiments [?].

5 Artificial Cell System

A membrane is an important structure for living systems. It distinguishes “self” from its environment and hierarchical structures inside the system (like cells, organs and so on) are composed by membranes. Membranes change their structure dynamically and constitute a system. We are interested in their dynamical structure in terms of computation.

A membrane is composed of “chemical compounds (denoted by symbols)” which are generated through chemical reactions in the cell. In each cell there is some chemical compounds and these chemical compounds interact with each other according to the rewriting rule (*reaction rules*).

Based on the principles outlined above, we develop an *“Artificial Cell System”* (*ACS*). It consists of a multiset of symbols, a set of rewriting rules (reaction rules) and membranes. *ACS* consists of an abstract chemical system, *“Abstract Rewriting System on MultiSets (ARMS)”* that is a multiset transform system,[19]. It consists of a multiset of symbols and a set of rewriting rules. Although not many alife researches have tackled this topic previously [10], [11], the focus of these researches is on the formation of a membrane. The aim of this study is to investigate the role of membrane in terms computation, thus we do not treat its formation. In this section, we introduce the basic structural ingredients of ARMS, membrane structures and how ACS works.

5.1 The membrane structure (MS)

To describe the membrane and its structure in *ARMS*, we first define the language MS over the alphabet $\{[,]\}$ whose strings are recurrently defined as follows:

1. $[,] \in MS$
2. if $\mu_1, ..., \mu_n \in MS$, n ≥ 1, then $[\mu_1, ..., \mu_n] \in MS$;
3. there is nothing else in MS.

The most outer membrane M_0 corresponds to a container such as a test tube or reactor and it never dissolves.

Consider now the following relation on MS: for $x, y \in MS$ we write $x \sim y$ if and only if we can write the two strings in the form $x = [_1...[_2...]_2[_3...]_3...]_1, y = [_1...[_3...]_3[_2...]_2...]$, i.e., if and only if two pairs of parentheses which are not contained in one other can be interchanged, together with their contents. We also denote by $\sim$ the reflexive and transitive closures of the relation $\sim$. This is clearly an equivalence relation. We denote by $\overline{MS}$ the set of equivalence classes of MS with respect to this relation. The elements of $\overline{MS}$ are called *membrane structures*.

It is easy to see that the parentheses [,] appearing in a membrane structure are matching correctly in the usual sense. Conversely, any string of correctly matching pairs of parentheses [,], with a matching pair at the ends, corresponds to a membrane structure.

Each matching pair of parentheses [,] appearing in a membrane structure is called a *membrane*. The number of membranes in a membrane structure μ is called the *degree* of μ and is denoted by deg(μ). The external membrane of a membrane structure μ is called the *vessel;* membrane of μ. When a membrane which appears in $\mu \in \overline{MS}$ has the form [ ] and no other membranes appear inside the two parentheses then it is called an *elementary* membrane.

5.2 ACS and ACSE

We will define two types of ACS;

1. ACS and
2. ACS with an Elementary membrane (ACSE).

ACSE is different only in the way of dissolving and dividing from ACS.

5.3 Descriptions of ACS

A transition ACS is a construct

$$\Gamma = (A, \mu, M_1, ..., M_n, R, MC, \delta, \sigma),$$

where:

1. A is a set of objects;
2. μ is a membrane structure (it can be changed throughout a computation);
3. $M_1, ..., M_n$, are multisets associated with the regions 1,2, ... n of μ;
4. R is a finite set of multiset evolution rules over A.
5. MC is a set of membrane compounds;

6. δ is the threshold value of dissolving a membrane;
7. σ is the threshold value of dividing a membrane;

μ is a membrane structure of degree n, n $\geq$ 1, with the membranes labeled in a one-to-one manner, for instance, with the numbers from 1 to n. In this way, also the regions of μ are identified by the numbers from 1 to n.

Rewriting rules are applied in following manner:

1. The same rules are applied to every membrane. There are no rules specific to a membrane.
2. All the rules are applied in parallel. In every step, all the rules are applied to all objects in every membrane that can be applied. If there are more than two rules that can apply to an object then one rule is selected randomly.
3. If a membrane dissolves, then all the objects in its region are left free in the region immediately above it.
4. All objects and membranes not specified in a rule and which do not evolve are passed unchanged to the next step.

Rewriting rule R is a finite set of multiset rewriting rules over A. Both the left and the right side of a rule are obtained by sampling with replacement of symbols. A set of reaction rules is constructed as the overall permutation of both sides of the rules.

Input and Output Chemical compounds are supported from outside of the system to M_0 and some compounds are exhausted from M_0. All chemical compounds are transformed among cells, a randomly selected chemical compound is transformed into the membrane just above or below it. Although a membrane does not allow specificity of transport across the membrane, a cell can control its chemical environment by chemical reaction.

5.3.1 Dissolving and dividing a membrane of ACS

A membrane is composed of a "membrane compound" which is in fact a symbol. To maintain a membrane, it needs to have a certain minimal volume. A membrane disappears if the volume of membrane compounds decreases below the needed volume to maintain the membrane. Dissolving the membrane is defined as follows:

$$[_h a, ...[_i b, ...]_i]_h \rightarrow [_h a, b, ...]_h,$$

where the ellipsis {...} illustrates chemical compounds inside the membrane. Dissolving takes place when

$$\frac{|w_i|_{MC}}{|M_i|} < \delta$$

where δ is a threshold value for dissolving the membrane. All chemical compounds in its region are then set free and they are merged into the region immediately above it.

On the other hand, when the volume of membrane compounds increases to a certain extent, then a membrane is divided. Dividing a membrane is realized by dividing it in multisets random sizes. The frequency at which a membrane is divided is decided in proportion to its size. As the size of a multiset becomes larger, the cell is divided more frequently. Technically, this is defined as follows;

$$[_h a, b, ...]_h \rightarrow [_h a, ...[_i b, ...]_i]_h$$

Dividing takes place when

$$\frac{|w_h|_{MC}}{|M_h|} > \sigma$$

where σ is a threshold for dividing the membrane. All chemical compounds in its region are then set free and they are separated randomly by new membranes.

5.4 Description of ACSE

ACSE is different only in the way of dividing and dissolving cells from ACS. Dissolving the membrane is defined as follows:

$$[_h a, b, ...]_h \rightarrow [_0 a, b, ...,]_0$$

Dissolving takes place when

$$\frac{|w_h|_{MC}}{|M_h} < \delta$$

where δ is a threshold value for dissolving the membrane. All chemical compounds in its region are then set free and they are merged into the region of M_0.

Dividing is defined as follows;

$$[_h a, b, ...]_h \rightarrow [_h a, ...]_h [_i b, ...]_i.$$

Dividing takes place when

$$\frac{|w_h|_{MC}}{|M_h|} > \sigma,$$

where σ is a threshold for dividing the membrane. All chemical compounds in its region are then set free and they are separated randomly in the old and new membranes. Hence, in ACSE, a structured cell such as $[a, b[c, [d, e]]]$ does not appear.

A Cell like Chemoton Because the components of a membrane diminish with the lapse of a certain time, a cell has to generate the components to maintain the membrane through chemical reactions in the cell in ACS and ACSE. Hence, all survived cells in ACS and ACSE become cells like a *chemoton*[5]. We confirmed this through simulations.

Evolution of Cells When a cell grows and the cell exceeds the threshold value for dividing, it divides into parts of random sizes. This can be seen as a kind of *mutation*. If a divided cell does not have any membrane compounds, it must disappear soon.

Furthermore, to maintain the membrane through chemical reactions inside the cell can be seen as *natural selection*. If cells can not maintain the membrane compounds, it must disappear soon.

Thus, both dividing membranes and dissolving membranes produce evolutionary dynamics. These correspondences are summarized into;

Natural Selection	Dissolving a membrane,
Mutation	Dividing a cell into parts of random size.

6 Behavior of *ACSE and ACS*

We will show some experimental results of ACSE and ACS in this section.

6.0.1 ACSE

The evolution of elementary cells can be regarded as an approximate model of the chemical evolution in the origin of life.

The following *ACSE* was simulated;

$\Gamma = (A = \{a, b, c\}, \mu = \{\ [,]_0, ...[,]_{100}\} M_0 = \{[a^{10}, b^{10}, c^{10}]^{100}\}, R, MC = \{b\}, \delta = 0.4, \sigma = 0.2)$,

where:

1. R, the length of the left- or right-hand-side of a rule is between one and three. Both sides of the rules are obtained by sampling with replacement of the three symbols a, b and c;

2. Membrane structures are assumed to be ($\mu = \{[_1]_1 ... [_{100}]_{100}\}$).

Through the simulation we discovered that the strength of a membrane affects the behavior of cells. The strength of a membrane is defined as the frequency of decreasing membrane compounds.

When a membrane is strong When a membrane is strong, the most stable cell consists of only one membrane, cells of this type become " mother" cells and they produce "daughter" cells.

In order to display a state of a cell we transform the state of a cell to a number by using the transformation function; $f(M(a), M(b), M(c)) = 10^2 \times M(a) + 10^1 \times M(b) + 10^0 \times M(c)$. For example, the state $\{a, a, b, c, c\}$ is transformed into $10^2 \times 2 + 10^1 \times 1 + 10^0 \times 2 = 212$.

The figure 2 illustrates the evolution of cells when a membrane is strong. The cells that are close to the horizontal axis are mother cells. Some daughter cells depart from the group and evolve different types of cells, even though almost all cells are in the group. In this case, dissolving a membrane compound takes place per 100 steps.

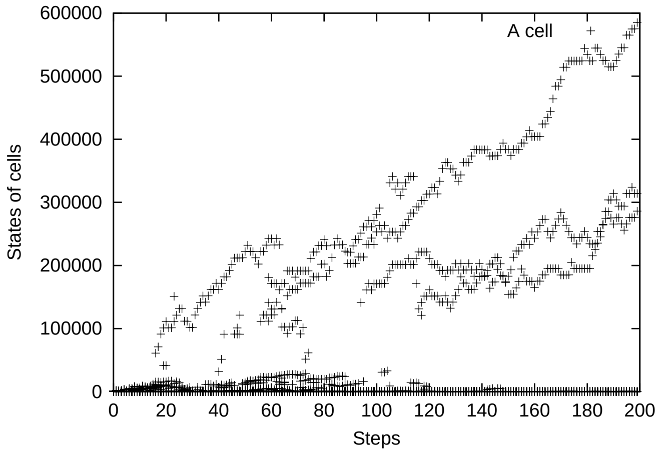


Figure 10: When a membrane is strong. The lines illustrate the regions where cells exist and points correspond to the state of cells.

The figure 3 is focused to the mother cells. At first there are about ten groups, and some of them become extinct: after 200 steps there remain about four groups.

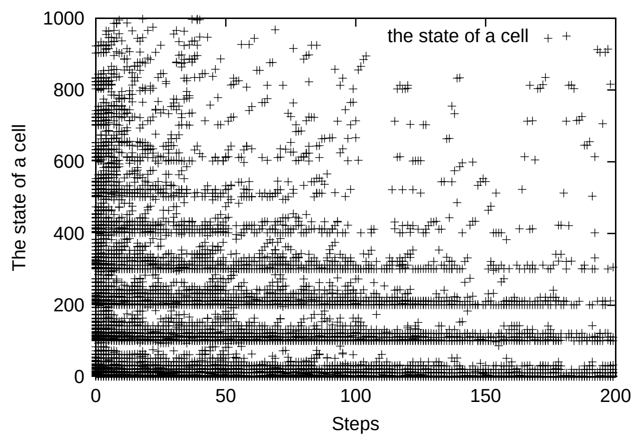


Figure 11: Evolution of mother cells. The points correspond to cells.

When a membrane is weak Figure 4 illustrates the case when a membrane is weak, a membrane dissolves every 3 steps. In this case, the system can not form a group of mother cells such as in the previous case. Since the group of cells drifts to more stable cells, the cells grow larger. Even if a large cell divides into parts of random sizes, the probability of including enough membrane compounds to maintain its membrane are larger than a small cell.

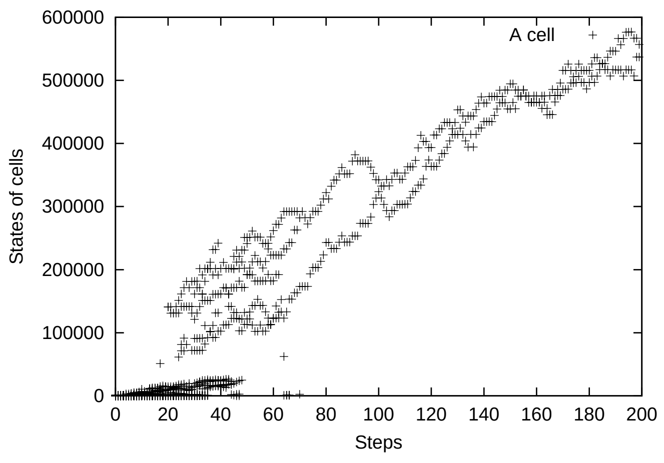


Figure 12: When a membrane is weak

We believe that this behaviors of evolution is similar to the evolution of viruses [20]. The settings of this simulation are so rough, however, that the possibility remains open that chemical evolution in origin of life is similar to virus evolution. This will be addressed in future research.

7 Genetic ACS (GACS)

Since a rewriting rule promotes a reaction, it can be regarded as an enzyme. Here we extend ACS with evolutionary mechanism We called the system Genetic ACS (GACS).

7.1 Descriptions of *GACS*

A transition in *GACS* is a construct

$$\Gamma = (A, \mu, M_1, ..., M_n, R, \delta, \sigma),$$

where:

1. A is a set of objects;
2. μ is a membrane structure (it can be changed throughout a computation);
3. $M_1, ..., M_n$, are multisets associated with the regions 1,2, ... n of μ;
4. R is a finite set of multiset evolution rules over A.
5. δ is the threshold of dissolving membrane;
6. σ is the threshold of dividing membrane.

The way of applying rewriting rules, the way of dissolving and dividing and input and output are the same as ACS and ACSE.

7.1.1 An enzyme

We denote a set of reaction rules as follows;

	a	b	c
a	x_{aa}	x_{ab}	x_{ac}
b	x_{ba}	x_{bb}	x_{bc}
c	x_{ca}	x_{cb}	x_{cc},

where x_{ij} means the number of compounds i which are transformed from j. For example, 2_{ab} means a rewriting rule, $b \rightarrow a, a$. We call the table a transformation table.

Transmission of an enzyme When a membrane is divided, the enzyme which is inside the membrane is copied and passed down to a new divided cell. At that time, a point mutation occurs only in the copied enzyme and it is passed down to the new cell. The enzyme remain in the old membrane as well as the new one. Point mutations occur every time a membrane divides. When a membrane is dissolved the enzyme which is inside the membrane loses its activity. A point mutation is a rewriting of the number of x_{ij}. Thus, it changes the number of transforming compounds to i. We assumed $x_{ij} \in \{0, 1, 2\}$. In ACS and ACSE the system have only one rewriting rule, however, in GACS, each membrane has a set of rules.

7.2 An experimental result of a GACS

We will show experimental results of GACS. At first, the following *GACS* was simulated;
$\Gamma = (A = \{a, b, c\}, \mu = \{\emptyset\} M_0 = \{a^{10}, b^{10}, c^{10}\}, R, MC = \{c\}, \delta = 0.4, \sigma = 0.2)$,

where the transformation table (R) is set as follows in the initial states;

	a	b	c
a	0_{aa}	0_{ab}	1_{ac}
b	1_{ba}	0_{bb}	0_{bc}
c	0_{ca}	1_{cb}	0_{cc}.

The productivity of membrane compounds p is defined as the ratio of the total number of non-membrane compounds to be produced to the total number of membrane compounds to be produced;

$$p = \frac{\Sigma_{j=a}^{j=b} x_{cj}}{\Sigma_{j=a}^{i=b} \Sigma_{j=a}^{j=c} x_{i,j}},$$

When $p = 0$ the enzyme does not produce any membrane compounds, when $p = 1$, it produces the same number of membrane compounds to the non- membrane compounds, and when $p > 1$, it produces more membrane compounds than non-membrane compounds. Figure 9 illustrates the time series of productivity, where the vertical axis illustrates the productivity, the horizontal axis illustrates the steps and each dot is an enzyme. It shows that at first almost enzymes evolve to $p > 1$. However, after 100 steps, the productivity of the enzymes decrease.

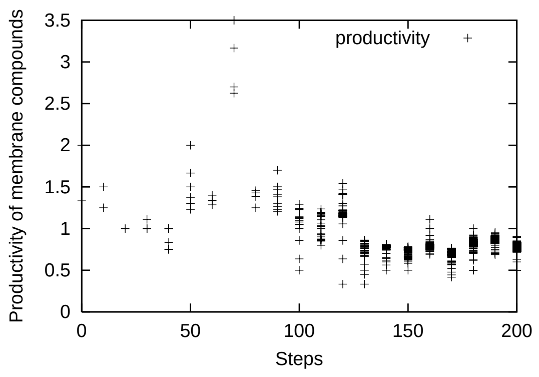


Figure 13: Productivity of enzymes.

After 100 steps, both the number of cells and the sizes of cells increase exponentially. Furthermore, the structure of cells becomes complicated. Figure 10 illustrates the correlation between the number of cells, the size of cells and the number of steps, where each dot corresponds to a cell.

Figure 11 illustrates the internal nodes of the whole system. If we regard M_0 as the root and other cells as internal nodes and leaves, we can regard the whole system as a tree. In order to indicate the complexity of the tree, we use the number of internal nodes in the tree. Figure 11 illustrates that the number of internal nodes increases exponentially, after 150 steps.

It is interesting that when a cell grows into a hierarchical cell, the enzyme evolves to a low productivity one.

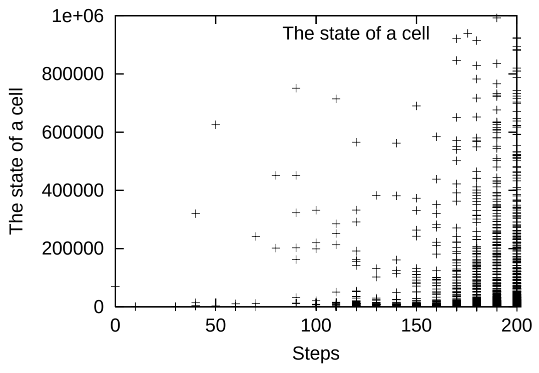


Figure 14: State transition of the system

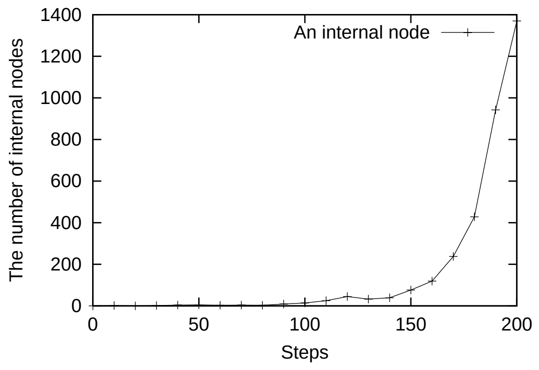


Figure 15: Internal nodes of the cell

The reason of this behavior can be considered as follows; the enzyme whose productivity is high always suffers from mutations, because it promotes membrane division thus it generates more mutations than low productivity ones. If every cell is an elementary cell, an enzyme have to keep producing membrane compounds at a high rate. However, when the cell forms structure, it is not necessary one with high productivity, because, in a structured cell, if an inside cell dissolves, the cell that includes the dissolved cell obtains its membrane compounds.

Therefore, during evolution of a cell into a structured cell, the cell needs a high productivity enzyme. However, once it evolves a structured cell, high productivity enzymes are weeded out. This is the role of membranes in terms of computation.

7.3 Genetic Programming by using GACS

We attempted to generate a program by using a GACS. In the GACS, a program corresponds to an enzyme, thus we breed an enzyme which can solve a particular problem. We apply it to a simple problem *doubling*; calculate the double value of the number of a and b then show the result as the number of c ($c = 2(a + b)$).

Description of GACS

A *GACS* is defined as follows;

A transition *GACS* is a construct $\Gamma = (A = \{a, b, c\}, \mu = \{[_1]_1 ... [_{100}]_{100}\}, M_0 = \{[a^2, b^2, c^0]^{100}\}, R)$. In the initial state, all transformation tables R are

	a	b	c
a	0_{aa}	0_{ab}	1_{ac}
b	1_{ba}	0_{bb}	0_{bc}
c	0_{ca}	1_{cb}	0_{cc},

and one hundred of elementary cells are assumed inside M_0. No compounds are transformed among cells and no input and output are assumed. Although we performed simulation by using different type of cells in the initial state, the experimental results are same, thus we will address only when each cell is $[a^2, b^2, c^0]$ in the initial state.

Dissolving and dividing a membrane The way of dissolving and dividing are the same as in ACSE. After n rewriting steps, if the number of c is smaller than 7, the membrane is dissolved and the enzyme inside it loses its activity, for example,

$$[_h a^3, b^5, c^8]_h [_i a^4, b^5, c^1]_i \rightarrow [_h a^3, b^5, c^8]_h.$$

In the above example, the number of c inside the membrane i is smaller than 7, so the membrane is dissolved.

After n rewriting steps, if the number of c is larger than 9, the membrane is divided and a point mutation takes place in its enzyme. Furthermore, a new enzyme is passed down to a new cell. When a cell divided, the inside multiset of the divided cell and its parent cell are set to $\{a^2, b^2, c^0\}$ again, so they try to solve the problem again. The results in:

$$[_h a^3, b^5, c^1 0]_h \rightarrow [_h a^2, b^2, c^0]_h [_i a^2, b^2, c^0]_i$$

In the example, because the number of c inside the membrane h is larger than 9, the membrane h is divided and a new membrane i emerges. Then compounds which are inside both membranes i and h are set to $\{a^2, b^2, c^0\}$. In this GACS, cells continue to solve the problem.

The fitness of the GACS The fitness of an enzyme is defined as the number of steps to reach the solution. By using this fitness, good enzymes are there that can solve the problem within a smaller number of steps than the others are selected.

7.3.1 Experimental result

At first, all enzymes are set to,

$$\begin{array}{ccc} a & \rightarrow & b, \\ b & \rightarrow & c, \\ c & \rightarrow & a. \end{array}$$

After 5000 rewriting steps, the enzymes that reach the solution within 8 steps are selected;

$$\begin{array}{ccc} a & \rightarrow & a, c, \\ b & \rightarrow & a, b, c, c, \\ c & \rightarrow & c, c; \end{array} \quad \begin{array}{ccc} a & \rightarrow & a, b, c, c, \\ b & \rightarrow & a, b, c, c, \\ c & \rightarrow & c, c; \end{array} \quad \begin{array}{ccc} a & \rightarrow & a, c, c, \\ b & \rightarrow & b, \\ c & \rightarrow & c, c. \end{array}$$

For each rule, one hundred of elementary cells that are $[a^2, b^2, c^0]$ are set again and calculations performed again. Next, the enzymes that can solve the problem within 5 steps are selected. Then, there remains only one enzyme;

$$\begin{array}{rcl} a & \rightarrow & a, c, c, \\ b & \rightarrow & c, c, \\ c & \rightarrow & c, c. \end{array}$$

it is a solution of this simulation. In fact a solution of this problem is

$$\begin{array}{rcl} a & \rightarrow & c, c, \\ b & \rightarrow & c, c, \\ c & \rightarrow & c; \end{array}$$

Thus the survived enzyme evolved to a similar enzyme in the solution.

By using this method, we have attempted to treat the system as an artificial living system of computation. In the future we plan to create a GACS as an artificial living system of computation that can solve more complicated problems. In such a system, in order to obtain the result, we observe their output (behaviors) and change the condition, do not stop their computations. In other words, we steer them in the right direction by using the selection and mutation, and lead them to our settled goal. Although GACS may not fit to make optimizer, we believe this method can apply to design artificial living things such as robots.

8 Conclusion

For the past, theoretical computer science has not been caused much attention in Life science. However, genome science is based on a huge strings of A,G,C,T and ALife study is based on computer science, from which we can predict that theoretical computer science will bring breakthroughs in Life science in the near future.

Acknowledgment

The authors would like to express many thanks to Dr. Gheorghe Păun for his useful comments, discussions and mathematical refinements. This research is supported by Grants-in Aid for Scientific Research No.11837005 from the Ministry of Education, Science and Culture in Japan and Inoue Foundation for Science.

References

[1] Dicke, M., Takabayashi, J., Shutte, C.,Krips, O. E. Behavioural ecology of plant-carnivore interactions: variation in response of phytoseiid mites to herbivore-induced plant volatiles. *Experimental and Applied Acarology 22*: 595-601, 1997

[2] Fontana, W. and L.W. Buss, The arrival of the fittest: Toward a theory of biological organization. *Bulletin of Mathematical Biology* 56: 1–64. 1994.

[3] Berry, G. and G. Boudol, The chemical abstract machine. *Theoretical Computer Science* 96: 217–248. 1992.

[4] Epstain, C. and R.Butow, Microarray technology -enhanced versatillity, persistent challaenge, Curr. Opin. Biotech, 11, 36-41, 2000

[5] Gánti, T., Organization of chemical reactions into dividing and metabolizing units: the chemotons. *Biosystems* 7: 189–195. 1975.

[6] Field, R. J. and M. Burger. 1985. *Oscillations and Traveling Waves in Chemical Systems.* John Wiley and Sons.

[7] Kauffuman, S., The Origins of Order, Oxford Univ. Press, 1993.

[8] Langton, C., Artificial Life, Artificial Life, pp1-47, Addison-Wewley, 1988.

[9] Maeda, T., Takabayashi, J.,Yano, S. and Takafuji, A. Factors affecting the resident time of the predatory mite Phytoseiulus persimilis (Acari: Phytoseiidae) in a prey patch *Applied Entomology and Zoology 33*: 573-576, 1998

[10] McMullin, B. and F. Varela, Rediscovering Computational Autopoiesis, *ECAL'97.* 1997.

[11] Mirazo, K., A. Moreno, F. Moran, et. al., Designing a Simulation Model of a Self-Maintaining Cellular System *ECAL'97.* 1997.

[12] Nicolis, G. and I. Prigogine. *Exploring Complexity, An Introduction.* San Francisco: Freeman and Company. 1989.

[13] Păun, G., Computing with Membranes, *Turku Center for Computer Science TUCS Technical Report No. 208* (submitted, also on http://www.tucs.fi). 1998.

[14] Păun, G., P Systems with Active Membranes: Attacking NP Complete Problems, Center for Discrete Mathematics and Theoretical Computer Science CDMTCS-102 (also on http://www.cs.auckland.ac.nz/CDMTCS). 1999.

[15] Sabelis, M. W., and M.,C.M. de Jong, 1988. Should all plants recruit bodyguards? Conditions for a polymorphic ESS synomone production in plants *Oikos*, 53 247-252,

[16] Smogyl, R, et.al., Cluster Analysis and data visualization of largescale gene expression data, Pasific Symposium on Biocomputing,3,42-53,1999.

[17] Suzuki, Y. and H. Tanaka. Order parameter for a Symbolic Chemical System, *Artificial Life VI*:130-139, MIT press. 1998.

[18] Suzuki, Y. and H. Tanaka. 1998. Symbolic chemical system based on abstract rewriting system and its behavior pattern. *Journal of Artificial Life and Robotics.* In press.

[19] Suzuki, Y. S., Tsumoto and H. Tanaka. Analysis of Cycles in Symbolic Chemical System based on Abstract Rewriting System on Multisets, *Artificial Life V*: 522-528. MIT press. 1996.

[20] Tanaka, H., F. Ren, S. Ogishima, Evolutionary Analysis of Virus Based on Inhomogeneous Markov Model, ISMB'99, p 148, 1999.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 286 - 295.

Mathematics of Multisets

Apostolos Syropoulos
Department of Civil Engineering
Democritus University of Thrace
GR-671 00 Xanthi, GREECE
email: `apostolo@obelix.ee.duth.gr`

Abstract

This paper is an attempt to summarize most things that are related to multiset theory. We begin by describing multisets and the operations between them. We continue with a categorical approach to multisets. Next, we present fuzzy multisets and their operations. Finally, we present partially ordered multisets.

1 Introduction

Many fields of modern mathematics have been emerged by *violating* a basic principle of a given theory only because useful structures could be defined this way. For example, modern non-Euclidean geometries have been emerged by assuming that the *Parallel Axiom*[1] does not hold. Similarly, *multisets*[2] [Knu81, MW85, Yag86] have been defined by assuming that for a given set A an element x occurs a finite number of times. From a practical point of view multisets are very useful structures, since many phenomena related to computer science can be described with them.

There are three methods to define a set and we are recalling them now, since they will be heavily used in the rest of the text:

i) A set is defined by naming all its members (*the list method*). This method can be used only for finite sets. Set A, whose members are $a_1, a_2, \ldots, a_n$ is usually written as
$$A = \{a_1, a_2, \ldots, a_n\}.$$

ii) A set is defined by a property satisfied by its members (*the rule method*). A common notation expressing this method is
$$A = \{x|P(x)\},$$
where the symbol $|$ denotes the phrase "such that", and $P(x)$ designates a proposition of the form "x has the property P."

[1]Which can be stated as follows: Given a point P not incident with line m, there is exactly one line incident with P and parallel to m.

[2]Multisets are also known as "bags", but many consider this term too vulgar. . .

iii) A set is defined by a function, usually called the *characteristic function*, that declares which elements of X are members of the set and which are not. Set A is defined by its characteristic function, χ_A, as follows:

$$\chi_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases}$$

In what follows we present the definition of multisets and the basic operations between multisets. Then, we proceed with a categorical approach to multisets by defining categories of multisets. Next, we present fuzzy multisets and their operations. We finish by presenting pomsets and their basic operations.

2 Multisets and their Operations

Ordinary sets are composed of pairwise different elements, i.e., no two elements are the same. If we relax this condition, i.e., if we allow multiple but finite occurrences of any element, we get a generalization of the notion of a set which is called a *multiset*. A multiset over some set D can be viewed as a function $f : D \to \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers. Formally, a multiset can be defined as follows:

Definition 2.1 Let D be a set. A multiset over D is just a pair $\langle D, f\rangle$, where D is a set and $f : D \to \mathbb{N}$ is a function.

The previous definition is the characteristic function definition method for multisets.

Remark 2.1 Any ordinary set A is actually a multiset $\langle A, \chi_A\rangle$, where χ_A is its characteristic function.

Since, multisets are sets with multiple but finite occurrences of any element, one can define a multiset by employing the list method. In what follows we will employ the most suitable definition method for each case we encounter.

An important notion in set theory is the notion of a subset. Moreover, for ordinary sets there are certain operations one can perform between sets, such as set intersection, union, etc. We proceed with the definitions of the notion of the subset of multiset and the operations between multisets.

Definition 2.2 Suppose that $\mathcal{A} = \langle A, f\rangle$ is a multiset, then the subset B of A is called the *support* of $\mathcal{A}$ if for every x such that $f(x) > 0$ this implies that $x \in B$, and for every x such that $f(x) = 0$ this implies that $x \notin B$.

It is clear that the characteristic function of B can be specified as:

$$\chi_B(x) = \min(f(x), 1).$$

Example 2.1 Given the multiset $\mathcal{A} = \{a, a, a, b, c, c\}$, then its support is the set $\{a, b, c\}$.

Definition 2.3 Assume that $\mathcal{A} = \langle A, f\rangle$ and $\mathcal{B} = \langle A, g\rangle$ are two multisets, we say that $\mathcal{A}$ is a sub-multiset of $\mathcal{B}$, denoted $\mathcal{A} \subseteq \mathcal{B}$ if for all $a \in A$

$$f(a) \leq g(a).$$

$\mathcal{A}$ is called a *proper* sub-multiset of $\mathcal{B}$, denoted $\mathcal{A} \subset \mathcal{B}$, if in addition for some $a \in A$

$$f(a) < g(a).$$

Obviously, it follows that for any two multisets $\mathcal{A} = \mathcal{B}$ iff $\mathcal{A} \subseteq \mathcal{B}$ and $\mathcal{B} \subseteq \mathcal{A}$.

Definition 2.4 Let $\mathcal{A} = \langle A, f\rangle$ be a multiset, then $\mathcal{A}$ is the *empty* multiset if for all $a \in A$, $f(x) = 0$.

Definition 2.5 Suppose that $\mathcal{A} = \langle A, f\rangle$ is a multiset, then its cardinality, denoted $\text{card}(\mathcal{A})$, is defined as

$$\text{card}(\mathcal{A}) = \sum_{a\in A} f(a).$$

Assume that A is a set, then $\mathcal{P}^A$ is the set of all multisets which have A as their support set. Moreover, A is the smallest multiset in $\mathcal{P}^A$ in the sense that if $\mathcal{B} \in \mathcal{P}^A$, then

$$\text{card}(\mathcal{B}) \geq \text{card}(A).$$

We are now turning our attention to the operations between multisets. We define, in this order, the sum, the removal, the union and the intersection of two multisets.

Definition 2.6 Suppose that $\mathcal{A} = \langle A, f\rangle$ and $\mathcal{B} = \langle A, g\rangle$ are two multisets, then their sum, denoted $\mathcal{A} \uplus \mathcal{B}$, is the multiset $\mathcal{C} = \langle A, h\rangle$, where for all $a \in A$:

$$h(a) = f(a) + g(a).$$

It can be easily shown that the multiset sum operation has the following properties:

i) Commutative: $\mathcal{A} \uplus \mathcal{B} = \mathcal{B} \uplus \mathcal{B}$;

ii) Associative: $(\mathcal{A} \uplus \mathcal{B}) \uplus \mathcal{C} = \mathcal{A} \uplus (\mathcal{B} \uplus \mathcal{C})$;

iii) There exists a multiset, the null multiset $\emptyset$, such that $\mathcal{A} \uplus \emptyset = \mathcal{A}$.

It is important to note that there exists no inverse and multiset sum is not indepotent.

Definition 2.7 Suppose that $\mathcal{A} = \langle A, f\rangle$ and $\mathcal{B} = \langle A, g\rangle$ are two multisets, then the removal of multiset $\mathcal{B}$ from $\mathcal{A}$, denoted $\mathcal{A} \ominus \mathcal{B}$, is the multiset $\mathcal{C} = \langle A, h\rangle$, where for all $a \in A$:

$$h(a) = \max\Big(f(a) - g(a), 0\Big).$$

Definition 2.8 Suppose that $\mathcal{A} = \langle A, f \rangle$ and $\mathcal{B} = \langle A, g \rangle$ are two multisets, then their union, denoted $\mathcal{A} \cup \mathcal{B}$, is the multiset $\mathcal{C} = \langle A, h \rangle$, where for all $a \in A$:

$$h(a) = \max\Big(f(a), g(a)\Big).$$

Definition 2.9 Suppose that $\mathcal{A} = \langle A, f \rangle$ and $\mathcal{B} = \langle A, g \rangle$ are two multisets, then their intersection, denoted $\mathcal{A} \cap \mathcal{B}$, is the multiset $\mathcal{C} = \langle A, h \rangle$, where for all $a \in A$:

$$h(a) = \min\Big(f(a), g(a)\Big).$$

The following properties can be easily established as holding for union, intersection and sum of multisets:

i) Commutativity:

$$\begin{aligned} \mathcal{A} \cup \mathcal{B} &= \mathcal{B} \cup \mathcal{A} \\ \mathcal{A} \cap \mathcal{B} &= \mathcal{B} \cap \mathcal{A}; \end{aligned}$$

ii) Associativity:

$$\begin{aligned} \mathcal{A} \cup (\mathcal{B} \cup \mathcal{C}) &= (\mathcal{A} \cup \mathcal{B}) \cup \mathcal{C} \\ \mathcal{A} \cap (\mathcal{B} \cap \mathcal{C}) &= (\mathcal{A} \cap \mathcal{B}) \cap \mathcal{C}; \end{aligned}$$

iii) Idempotency:

$$\begin{aligned} \mathcal{A} \cup \mathcal{A} &= \mathcal{A} \\ \mathcal{A} \cap \mathcal{A} &= \mathcal{A}; \end{aligned}$$

iv) Distributivity:

$$\begin{aligned} \mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) &= (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{A} \cup \mathcal{C}) \\ \mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) &= (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C}); \end{aligned}$$

v)

$$\begin{aligned} \mathcal{A} \uplus (\mathcal{B} \cup \mathcal{C}) &= (\mathcal{A} \uplus \mathcal{B}) \cup (\mathcal{A} \uplus \mathcal{C}) \\ \mathcal{A} \uplus (\mathcal{B} \cap \mathcal{C}) &= (\mathcal{A} \uplus \mathcal{B}) \cap (\mathcal{A} \uplus \mathcal{C}); \end{aligned}$$

vi)

$$\begin{aligned} \mathcal{A} \cap (\mathcal{A} \uplus \mathcal{B}) &= \mathcal{A} \\ \mathcal{A} \cup (\mathcal{A} \uplus \mathcal{B}) &= \mathcal{A} \uplus \mathcal{B}; \end{aligned}$$

vii)

$$\mathcal{A} \uplus \mathcal{B} = (\mathcal{A} \cup \mathcal{B}) \uplus (\mathcal{A} \cap \mathcal{B}).$$

Let $\mathcal{A} = \langle X, f\rangle$ be a multiset and $B \subseteq X$. We are interested in forming the multiset $\mathcal{C} = \langle X, g\rangle$, where

$$\begin{array}{rcll} g(x) & = & f(x) & \text{if } x \in B \\ g(x) & = & 0 & \text{if } x \notin B \end{array}$$

or, in other words, $g(x) = f(x) \cdot \chi_B(x)$.

A closely related problem is that of forming a multiset by removing all the elements from $\mathcal{A}$ which are in the set B. That is, we are interested in forming the multiset $\mathcal{D} = \langle X, h\rangle$, where

$$\begin{array}{rcll} h(x) & = & 0 & \text{if } x \in B \\ h(x) & = & f(x) & \text{if } x \notin B \end{array}$$

which can be expressed compactly as follows:

$$h(x) = f(x) \cdot (1 - \chi_B(x)), \forall x \in B.$$

However, one may note that $1 - \chi_B(x)$ is the characteristic function of the *complement* set of B, denoted $\bar{B}$. So, the previous equation becomes

$$h(x) = f(x) \cdot \chi_{\bar{B}}(x), \forall x \in B.$$

thus

$$\mathcal{E} = \mathcal{A} \circledast \bar{B}.$$

We shall call the operation $\mathcal{A} \circledast B$ *multi-intersection.* In general, it holds that if $\mathcal{A} = \langle X, \chi_A\rangle$, $A \subseteq X$, and B is a set, then $\mathcal{A} \circledast B = A \cap B$. Moreover, the following properties do hold:

$$\begin{array}{rcl} A \circledast X & = & A \\ A \circledast \emptyset & = & \emptyset \\ (\mathcal{A}_1 \cap \mathcal{A}_2) \circledast B & = & (\mathcal{A}_1 \circledast B) \cap (\mathcal{A}_2 \circledast B) \\ (\mathcal{A}_1 \cup \mathcal{A}_2) \circledast B & = & (\mathcal{A}_1 \circledast B) \cup (\mathcal{A}_2 \circledast B) \end{array}$$

3 Categorical Models of Multisets

This section assumes that the reader is familiar with the basics of Category Theory. Readers not familiar with Category Theory should consult any good book on the subject such as [BW99, McL98, vO95].

Let $\mathcal{C}$ be a category. A functor $E : \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ is called a *presheaf* on $\mathcal{C}$. Thus a presheaf on $\mathcal{C}$ is a contravariant functor. The presheaves on $\mathcal{C}$ with natural transformations as arrows form a category denoted $\mathbf{Psh}(\mathcal{C})$. Presheaves are used to categorically define multisets:

Definition 3.1 Let A be a set treated as a discrete category. A functor $F : A \to \mathbf{Set}$ is a multiset of elements of A. If $a \in A$, the set $F(a)$ denotes the multiplicity to which a occurs in A.

In [Tay89] Paul Taylor gives the following somehow more general definition:

Definition 3.2 Let A be a set. A multiset of A is an assignment of an abstract set X_a (its **multiplicity**) to each element $a \in A$. Abstractly, a multiset is represented by a multiplicity function $X_\_ : A \to \mathbf{Set}$ or by a display $x : X = (\bigcup_{a \in A} X_a) \to A$. Hence a finite multiset may be written as an unordered list (with repetition) whose term are from A.

Based on this definition, we can define the category of multisets of A as follows:

Definition 3.3 $\mathbf{Psh}(A)$ denotes the category of multisets of A. There are three ways of seeing its arrows: (α) as an A-indexed family of functions $f_a : X_a \to Y_a$; (β) as a natural transformation $f : X_\_ \to Y_\_$ between functors $A \rightrightarrows \mathbf{Set}$, or ($\gamma$) as functions $f : X \to Y$ such that $y \circ f = x$.

Arrows between multisets in the category $\mathbf{Psh}(A)$ are "color-preserving" function. One problem that remains is the definition of a category of all possible multisets.

Definition 3.4 **MSet** is a big category whose objects are pairs $(A, \mathbf{Psh}(A))$. An arrow from $(A, \mathbf{Psh}(A))$ to $(B, \mathbf{Psh}(B))$ is a pair (F, α) where F is a functor from A to B and α is functor from $\mathbf{Psh}(A)$ to $\mathbf{Psh}(B)$.

Another definition of the category of all possible multisets due to Ieke Moerdijk [Moe00] is the following one:

Definition 3.5 $\mathbf{MSet}_2$ is a big category whose objects are pairs (A, P), where A is a set treated as a discrete category and P a presheaf on A. Given two objects (A, P) and (B, Q) an arrow between them is a pair $(\mathcal{F}, \mathtt{F})$, where $\mathcal{F} : A \to B$ is a functor and $\mathtt{F} : P \to Q$ is presheaf map. Moreover, $P = Q \circ \mathcal{F}$.

4 Fuzzy Multisets

Fuzzy set theory has been introduced as a means to deal with vagueness in mathematics. The theory is well-established and we will not get into the trouble of presenting it. We just note that fuzzy set theory was an attempt to develop a formal apparatus to involve a partial membership in a set, mainly to arm people in the modeling of empirical objects and facts. In other words, fuzzy set theory is, sort to say, a generalization of the notion of set membership.

Definition 4.1 Suppose that X is a set, any function $A : X \to \mathrm{I}$, where $\mathrm{I} = [0, 1]$, is called a fuzzy subset of X. Function A is usually called the membership function of the fuzzy subset A.

Fuzzy multisets have been introduced by Yager [Yag86] and have been studied by Miyamoto [Miy96, Miy99] and others. A fuzzy multiset of some set X is just a multiset of $X \times \mathrm{I}$. We are now defining summation of fuzzy multisets:

Definition 4.2 Assume that $\mathcal{A} = \langle X \times \mathrm{I}, f\rangle$ and $\mathcal{B} = \langle X \times \mathrm{I}, g\rangle$ are two fuzzy multisets, then their sum, denoted $\mathcal{A} \uplus \mathcal{B}$, is the fuzzy multiset $\mathcal{C} = \langle X \times \mathrm{I}, h\rangle$, where for all $(x, \mu_x) \in X \times \mathrm{I}$:

$$h(x, \mu_x) = f(x, \mu_x) + g(x, \mu_x).$$

As in the case of *crisp*[3] multisets there is more that one way to define a fuzzy multiset. In order to define the basic operation between fuzzy multisets we define fuzzy multisets by the list method. Let $\mathcal{A} = \{(x_i, \mu_i)\}_{i=1,..,p}$ be a fuzzy multiset, then we can write the same set as $\mathcal{A} = \{\{\mu_{11}, \ldots, \mu_{1\ell_1}\}/x_1, \ldots, \{\mu_n, \ldots, \mu_{n\ell_n}\}/x_n\}$. Note that $\{\mu_{11}, \ldots, \mu_{1\ell_1}\}$ is actually a multiset of I. Next, we rearrange the multisets $\{\mu_{11}, \ldots, \mu_{1\ell_1}\}$ so all elements appear in decreasing order. Finally, we need to add zeroes so that the length of all multisets $\{\mu_{11}, \ldots, \mu_{1\ell_1}\}$ is the same. This representation is called the *graded sequence*. To make things clear we give an example:

Example 4.1 Let

$$\begin{aligned}\mathcal{A} &= \{(a, 0.2), (b, 0.5), (b, 0.1),\\ &\quad (a, 0.2), (a, 0.3), (d, 0.7)\}\end{aligned}$$

be a fuzzy multiset, then its graded sequence follows:

$$\begin{aligned}\mathcal{A} &= \{\{0.3, 0.2, 0.2\}/a, \{0.5, 0.1, 0\}/b,\\ &\quad \{0.7, 0, 0\}/d\}\end{aligned}$$

In case we want to perform certain operations on two or more fuzzy multisets, all multisets $\{\mu_{11}, \ldots, \mu_{1\ell_1}\}$ must have the same length. Moreover, even if one fuzzy multiset does not contain an element c we must add an entry of the form $\{\underbrace{0, 0, \ldots, 0}_{p \text{ times}}/c\}$, where p is the length of all other multisets. We are now giving the definitions of the various operations between fuzzy multisets:

Definition 4.3 Assume that $\mathcal{A} = \{\{\mu_{1p}, \ldots, \mu_{11}\}/x_1, \ldots, \{\mu_{np}, \ldots, \mu_{n1}\}/x_n\}$ and $\mathcal{B} = \{\{\mu'_{1p}, \ldots, \mu'_{11}\}/x_1, \ldots, \{\mu'_{np}, \ldots, \mu'_{n1}\}/x_n\}$ are two fuzzy multisets, then

i) $\mathcal{A} \subseteq \mathcal{B}$ iff for every x_i, $\mu_{ij} \le \mu'_{ij}$, $j = 1, \ldots, p$.

ii) $\mathcal{A} = \mathcal{B}$ iff for every x_i, $\mu_{ij} = \mu'_{ij}$, $j = 1, \ldots, p$.

iii) $\mathcal{C} = \mathcal{A} \cup \mathcal{B}$, where $\mathcal{C} = \{\{\mu''_{1p}, \ldots, \mu''_{11}\}/x_1, \ldots, \{\mu''_{np}, \ldots, \mu''_{n1}\}/x_n\}$ iff for every x_i, $\mu''_{ij} = \max(\mu'_{ij}, \mu_{ij})$, $j = 1, \ldots, p$.

iv) $\mathcal{C} = \mathcal{A} \cap \mathcal{B}$, where $\mathcal{C} = \{\{\mu''_{1p}, \ldots, \mu''_{11}\}/x_1, \ldots, \{\mu''_{np}, \ldots, \mu''_{n1}\}/x_n\}$ iff for every x_i, $\mu''_{ij} = \min(\mu'_{ij}, \mu_{ij})$, $j = 1, \ldots, p$.

In case functions max and min are replaced by a t-norm t and a t-conorm s respectively, we obtain the definitions for $\cap_t$ and $\cup_s$, respectively. The union and intersection of arbitrary fuzzy multisets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ satisfy the following laws:

[3]In fuzzy set theory the term crisp is used to characterize anything that is non-fuzzy.

i) Commutative law.

$$\begin{aligned}\mathcal{A}\cup\mathcal{B} &= \mathcal{B}\cup\mathcal{A}\\ \mathcal{A}\cap\mathcal{B} &= \mathcal{B}\cap\mathcal{A}\end{aligned}$$

ii) Associative law.

$$\begin{aligned}(\mathcal{A}\cup\mathcal{B})\cup\mathcal{C} &= \mathcal{A}\cup(\mathcal{B}\cup\mathcal{C})\\ (\mathcal{A}\cap\mathcal{B})\cap\mathcal{C} &= \mathcal{A}\cap(\mathcal{B}\cap\mathcal{C})\end{aligned}$$

iii) Distributive law.

$$\begin{aligned}(\mathcal{A}\cup\mathcal{B})\cap\mathcal{C} &= (\mathcal{A}\cap\mathcal{B})\cup(\mathcal{A}\cap\mathcal{C})\\ (\mathcal{A}\cap\mathcal{B})\cup\mathcal{C} &= (\mathcal{A}\cup\mathcal{B})\cap(\mathcal{A}\cup\mathcal{C})\end{aligned}$$

Next, we define the α-cut for fuzzy multisets. We first recall what the α-cut of an ordinary fuzzy set is:

Definition 4.4 Let U be a set, let C be a partially ordered set and let $A : U \to C$. For $\alpha \in C$, the α-cut of A, is $A^{-1}(\uparrow\alpha) = \{u \in U : A(u) \geq \alpha\}$. The subset of U will be denoted by A_α.

We proceed with the definition of the α-cut of a fuzzy multiset:

Definition 4.5 Assume that $\mathcal{A} = \langle X \times \mathrm{I}, f\rangle$ is a fuzzy multiset, and that $\alpha \in (0, 1]$, then $\mathcal{A}_\alpha = \langle X, f'\rangle$, i.e., the α-cut of $\mathcal{A}$, is a multiset such that

$$f'(x) = \sum_{\mu_x \geq \alpha} f(x, \mu_x).$$

Consequently, the α-cut of a fuzzy multiset is just a multiset.

Given a fuzzy multiset $\mathcal{A} = \langle X \times \mathrm{I}, h\rangle$ and function $f : X \to Y$ we can define two images:

$$\begin{aligned}f[\mathcal{A}] &= \uplus_{x\in\mathcal{A}}\{f(x)\}\\ f(\mathcal{A}) &= \bigcup_{x\in\mathcal{A}}\{f(x)\}\end{aligned}$$

Note that in case the fuzzy multiset is just a fuzzy subset, the second images corresponds to the *extension principle* of fuzzy set theory.

5 Partially Ordered Multisets

Partially ordered multisets (or just pomsets) have been used by Pratt [Pra86] as a means to model concurrency. In this model a process is a set of pomsets. Here we will only present the definition of a pomset and the basic operations between pomsets. The reader interested in learning more on their use on modeling concurrency is refereed to Pratt's paper. The following definition of pomset is due to Gischer[Gis84] and is copied verbatim from Pratt's paper:

Definition 5.1 A *labeled partial order* (lpo) is a 4-tuple $(V, \Sigma, \leq, \mu)$ consisting of

i) a *vertex set* V, typically modeling *events*;

ii) an *alphabet* Σ (for symbol set), typically modeling *actions* such as the arrival of integer 3 at port Q;

iii) a *partial order* $\leq$ on V, with $e \leq f$ typically being interpreted as event e necessarily preceding event f in time; and

iv) a *labeling function* $\mu : V \to \Sigma$ assigning symbols to vertices, each labeled event representing an *occurrence* of the action labeling it, with the same action possibly having multiply occurrences, that is, μ need not be injective.

A *pomset* is then the isomorphism class of an lpo, denoted $[V, \Sigma, \leq, \mu]$.

Now we are ready to define the basic operations between pomsets:

Definition 5.2 Assume that $p = [V, \Sigma, \leq, \mu]$ and $p' = [V', \Sigma', \leq', \mu']$ are two pomsets, then:

i) their *concurrence* $p||p'$ is the pomset $[V \cup V', \Sigma \cup \Sigma', \leq \cup \leq', \mu \cup \mu']$, where V and V' are assumed to be disjoint;

ii) their *concatenation* $p;p'$ is as for concurrence except that instead of $\leq \cup \leq'$ the partial order is taken to be $\leq \cup \leq' \cup (V \times V')$; and

iii) their *orthocurrence* $p \times p'$ is the pomset $[V \times V', \Sigma \times \Sigma', \leq \times \leq', \mu \times \mu']$.

References

[BW99] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Les Publication CRM, Montréal, third edition, 1999.

[Gis84] J. Gischer. *Partial Orders and the Axiomatic Theory of Shuffle.* PhD thesis, Computer Science Dept., Stanford University, 1984.

[Knu81] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addisson-Wesley, 1981.

[McL98] Saunders McLane. *Category Theory for the Working Mathematician.* Springer-Verlag, New York, second edition, 1998.

[Miy96] Sadaaki Miyamoto. Fuzzy multisets and application to rough approximation of fuzzy sets. In *Proceedings of the Fourth International Workshop on Rough Sets Fuzzy Sets, and Machine Discovery (RSFD'96)*, pages 255–260, 1996.

[Miy99] Sadaaki Miyamoto. Two images and two cuts in fuzzy multisets. In *Proc. of the 8th International Fuzzy Systems Association World Congress (IF-SA'99)*, pages 1047–1051, 1999.

[Moe00] Ieke Moerdijk. Personal communication, 2000.

[MW85] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming*, volume 1 Deductive Reasoning. Addison-Wesley, Reading, Massachusetts, 1985.

[Pra86] Vaughan Pratt. Modellin Concurrenct with Partial Orders. *Int. J. of Parallel Programming*, 15(1):33–71, 1986.

[Tay89] P. Taylor. Quantitative domains, groupoids and linear logic. In D. Pitt, D. Rydeheard, P. Dybjer, A. Pitt, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[vO95] Jaap van Oosten. Basic Category Theory. Technical Report LS-95-1, BRICS, January 1995.

[Yag86] Ronald R. Yager. On the theory of bags. *Int. J. General Systems*, 13:23–37, 1986.

Pre-proceedings of the Workshop on Multiset Processing (Curtea de Arges, August 21-25, 2000), pages 296 - 311.

Using Membrane Features in P Systems*

C. Zandron, C. Ferretti, G. Mauri

DISCO - Universitá di Milano-Bicocca - Italy

Abstract

In the basic variant of P systems, membranes are used as separators and as channels of communication. Other variants, introduced to obtain more "realistic" models, consider membranes with different features: membranes of variable thickness, electrically charged membranes and active membranes (membranes can be divided in two or more membranes). These features are not only useful to obtain "realistic" models: we show how we can use them to get simpler and faster models.

1 Introduction

The P systems were recently introduced in [6] as a class of distributed parallel computing devices of a biochemical type.

The basic model consists of a membrane structure composed by several cell-membranes, hierarchically embedded in a main membrane called the skin membrane. The membranes delimit regions and can contain objects. The objects evolve according to given evolution rules associated with the regions. A rule can modify the objects and send them outside the membrane or to an inner membrane. Moreover, the membranes can be dissolved. When a membrane is dissolved, all the objects in this membrane remain free in the membrane placed immediately outside, while the evolution rules of the dissolved membrane are lost. The skin membrane is never dissolved.

The evolution rules are applied in a maximally parallel manner: at each step, all the objects which can evolve should evolve. A computation device is obtained: we start from an initial configuration and we let the system evolve. A computation halts when no further rule can be applied. The objects in a specified output membrane are the result of the computation.

In this basic variant, the membranes are used only as separators of objects and as channels of communication. In [7] and [8] new features are introduced:

- **Membranes of variable thickness:** membranes can be made thicker or thinner (also dissolved as said before). Initially, all membranes have thickness 1. If a membrane has thickness 2, then no object can pass through it.

*This work has been supported by the Italian Ministry of University (MURST), under project "Unconventional Computational Models: Syntactic and Combinatorial Methods".

- **Membranes with electrical charges:** electrical charges are associated to membranes and to objects: they can be marked with "positive" (+), "negative" (−) or "neutral" (0). The charge define the communication of the objects: an object marked with + (respectively −) will enter a membrane marked with − (respectively +), nondeterministically chosen from the set of membranes adjacent to the region where the object is produced. The neutral objects are not introduced in an inner membrane. Thus, the evolution rules do not specify the label of the membrane where the object will be sent, but they only associate a charge to every object involved in the application of the rule itself.
- **Active membranes:** the membranes can not only be dissolved, but they can multiply by division. We consider here division rules for elementary membranes only, i.e. membranes not containing other membranes (in [8] one considers division for non-elementary membranes too). Starting from a membrane we get two (or more) membranes, each with the same objects and rules of the original membrane.

Other variants are considered in [1], [6], [9] and [10].

In this paper we show how to use the previous features to get simpler and faster models of P systems. In particular we show how to:

- Use variable thickness to simulate priority in (Rewriting) P Systems.
- Use electrical charges to simplify membrane structures in (Rewriting) P systems.
- Use membrane division for elementary membranes to execute fast computations.

2 P systems and Rewriting P systems

We give a very short definition of P system; details and examples can be found in [5], [6] and [7]. We refer to [13] for elements of Formal Language Theory.

A membrane structure is a construct consisting of several membranes placed in a unique membrane; this unique membrane is called skin membrane. We identify a membrane structure with a string of correctly matching parentheses, placed in a unique pair of matching parentheses; each pair of matching parentheses corresponds to a membrane.

A membrane identifies a region, delimited by it and the membrane immediately inside it. If in the regions we place multi sets of objects from a specified finite set V, we get a super-cell.

A super-cell system (or P system) is a super-cell provided with evolution rules for its objects and with a designated output membrane.

Such a system of degree $n, n \geq 1$, is a construct

$$\Pi = (V, \mu, M_1, \ldots, M_n, (R_1, \rho_1), \ldots, (R_n, \rho_n), i_0)$$

where:

- V is an alphabet
- μ is a membrane structure consisting of n membranes (labeled with $1, ..., n$)
- $M_i, 1 \leq i \leq n$ are multi sets over V associated with the regions $1, 2, ..., m$ of μ;
- $R_i, 1 \leq i \leq n$ are finite sets of evolution rules associated with the regions $1, 2, ..., m$ of μ; ρ_i is a partial order relation over R_i, $1 \leq i \leq m$, specifying a priority relation among rules of R_i. The rules are of the form $u \rightarrow v$ where u is a symbol of V and $v = v^{'}$ or $v = v^{'}\delta$. $v^{'}$ is a string over $(V \times \{here, out\}) \cup (V \times \{in_j | 1 \leq j \leq m\})$ and δ is a special symbol not in V. When we apply a rule containing the symbol δ, the membrane where the rule is applied is dissolved. The rules of that membrane are deleted and the objects remain free in the membrane placed immediatly outside. The skin membrane is never dissolved.
- i_0 is the output membrane. If we don't specify the output membrane, we consider as output the objects sent out from the skin membrane in the order of their expelling from the system (this variant was introduced in [9]).

We consider here not only P systems but *Rewriting P Systems* (RP Systems) too. In such systems, objects can be described by finite strings over a given finite alphabet. The evolution of an object will correspond to a transformation of the strings. Consequently, the evolution rules are given as rewriting rules. We only use context-free rewriting rules.

We describe now three variants of P systems, which consider membranes with different features:

- **Membranes of variable thickness:** the rules are of the form $u \rightarrow v(tar)$ where u is a symbol of V and $v = v^{'}$ or $v = v^{'}\delta$ or $v = v^{'}\tau$. $v^{'}$ is a string over V, while δ, τ are special symbols not in V. $tar \in \{here, out, in_m\}, 1 \leq m \leq n$ represents the target membrane, i.e. the membrane where the string produced with this rule will go.

  If a rule contains the special symbol δ and the membrane where this rule is applied has thickness 1, then that membrane is dissolved and it is no longer recreated; the objects in the membrane become objects of the membrane placed immediately outside, while the rules of the dissolved membrane are removed. If the membrane has thickness 2, this symbol reduces the thickness to 1. If a rule contains the special symbol τ the thickness of the membrane where this rule is applied is increased; the thickness of a membrane of thickness 2 is not further increased. If a membrane has thickness 2, then no object can pass through it. If both the symbols δ and τ are introduced in the same region, the corresponding membrane preserves its thickness.

  The communication of objects has priority on the actions of δ and τ; if at the same step an object has to pass through a membrane and a rule changes the thickness of that membrane, then we first transmit the object and after that we change the thickness.

- **Membranes with electrical charges:** μ is a membrane structure consisting of n membranes; each membrane is marked with one of the symbols $+, -, 0$.

  The rules are of the form

  $$(u \rightarrow v(p))$$

  where u is a symbol of V and $v = v^{'}$ or $v = v^{'}\delta$. $v^{'}$ is a string over V, δ is a special symbol not in V and $p \in \{here, out, +, -\}$.

  A rule can marks the object with $+, -$, *out, here.* If a rule marks a string with *here* (or if the mark is omitted), it means that the string obtained after the rule is applied will remain in the same region where the rule is applied. If the mark is *out*, the string will be sent to the region placed immediately outside. If the string is marked with $+$ (or $-$), it will be sent through a membrane marked with $-$ (respectively $+$) and adjacent to the region where the rule is applied.

- **Active (and electrically charged) membranes:** A P system with active membranes is a construct $\Pi = (V, T, H, \mu, w_1, \ldots, w_m, R)$ where:

  - $m \geq 1$
  - V is an alphabet
  - $T \subseteq V$ is the terminal alphabet
  - H is a finite set of labels for membranes
  - μ is a membrane structure consisting of m membranes, labeled (not necessarily in a one-to-one manner) with elements of H; all membranes in μ are supposed to be neutral
  - $w_1, \ldots, w_m$ are strings over V, describing the multisets of objects placed in the m regions of μ
  - R is a finite set of developmental rules, of the following forms:

  1. Type (a): $[_h a \rightarrow v]_h^{\alpha}$, for $h \in H, a \in V$, $v \in V^*$, $\alpha \in \{+, -, 0\}$ (object evolution rules),
  2. Type (b): $a[_h]_h^{\alpha_1} \rightarrow [_h b]_h^{\alpha_2}$, where $a, b \in V$, $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$ (an object is introduced in membrane h),
  3. Type (c): $[_h a]_h^{\alpha_1} \rightarrow [_h]_h^{\alpha_2} b$, for $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in V$ (an object is sent out from membrane h),
  4. Type (d): $[_h a]_h^{\alpha} \rightarrow b$, for $h \in H$, $\alpha \in \{+, -, 0\}$, $a, b \in V$ (membrane h is dissolved),
  5. Type (e): $[_h a]_h^{\alpha_1} \rightarrow [_h b]_h^{\alpha_2} [_h c]_h^{\alpha_3}$, for $h \in H$, $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$, $a, b, c \in V$ (division rules for elementary membranes)
  6. Type (f): $[_{h_0} [_{h_1}]_{h_1}^{\alpha_1} \ldots [_{h_k}]_{h_k}^{\alpha_1} [_{h_{k+1}}]_{h_{k+1}}^{\alpha_2} \ldots [_{h_n}]_{h_n}^{\alpha_2}]_{h_0}^{\alpha_0} \rightarrow$
     $[_{h_0} [_{h_1}]_{h_1}^{\alpha_3} \ldots [_{h_k}]_{h_k}^{\alpha_3}]_{h_0}^{\alpha_5} [_{h_0} [_{h_{k+1}}]_{h_{k+1}}^{\alpha_4} \ldots [_{h_n}]_{h_n}^{\alpha_4}]_{h_0}^{\alpha_6}$,
     for $k \geq 1, n > k, h_i \in H, 0 \leq i \leq n$, and $\alpha_0, ..., \alpha_6 \in \{+, -, 0\}$ with $\{\alpha_1, \alpha_2\} = \{+, -\}$ (division rules for non-elementary membranes)

The rules are applied following the principles in [8]. When a membrane is divided by a rule of type (e) or (f), then the content of this membrane is reproduced unchanged in the new copies we get.

In the next chapters we will need the notion of matrix grammar, too. Such a grammar is a construct $G = (N, T, S, M, C)$, where N, T are disjoint alphabets, $S \in N$, M is a finite set of sequences of the form $(A_1 \to x_1, ..., A_n \to x_n)$, $n \geq 1$, of context-free rules over $N \cup T$ (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases), and C is a set of occurrences of rules in M (N is the nonterminal alphabet, T is the terminal alphabet, S is the axiom, while the elements of M are called matrices). For $w, z \in (N \cup T)^*$ we write $w \Rightarrow z$ if there is a matrix $(A_1 \to x_1, ..., A_n \to x_n)$ in M and the strings $w_i \in (N \cup T)^*, 1 \leq i \leq n+1$, such that $w = w_1, z = w_{n+1}$, and, for all $1 \leq i \leq n$, either $w_i = w_i^{'} A_i w_i^{''}, w_{i+1} = w_i^{''} x_i w_i^{''}$, for some $w_i^{'}, w_i^{''} \in (N \cup T)^*$, or $w_i = w_{i+1}, A_i$ does not appear in w_i, and the rule $A_i \to x_i$ appears in C (the rules of a matrix are applied in order, possibly skipping the rules in C if they cannot be applied; we say that these rules are applied in the appearance checking mode.) If $C = \emptyset$ then the grammar is said to be without appearance checking (and C is no longer mentioned). We denote by $\Rightarrow^*$ the reflexive and transitive closure of the relation $\Rightarrow$. The language generated by G is defined by $L(G) = \{w \in T^* | S \Rightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac}. When we use only grammars without appearance checking, then the obtained family is denoted by MAT.

A matrix grammar $G = (N, T, S, M, C)$ is said to be in the binary normal form if $N = N_1 \cup N_2 \cup \{S, \dagger\}$, with these three sets mutually disjoint, and the matrices in M are of one of the following forms:

1. $(S \to XA)$, with $X \in N_1, A \in N_2$,
2. $(X \to Y, A \to x)$, with $X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*$,
3. $(X \to Y, A \to \dagger)$, with $X, Y \in N_1, A \in N_2$,
4. $(X \to \lambda, A \to x)$, with $X \in N_1, A \in N_2$, and $x \in T^*$.

Moreover, there is only one matrix of type 1 and C consists exactly of all rules $A \to \dagger$ appearing in matrices of type 3. One sees that $\dagger$ is a trap-symbol; once introduced, it is never removed. A matrix of type 4 is used only once, at the last step of a derivation (clearly, matrices of forms 2 and 3 cannot be used at the last step of a derivation). According to Lemma 1.3.7 in [6], for each matrix grammar there is an equivalent matrix grammar in the binary normal form.

We denote by CF and RE the families of context-free and recursively enumerable languages respectively. It is known that $CF \subset MAT \subset MAT_{ac} = RE$. Further details about Matrix grammars can be found in [2] and in [13]. Moreover, in [3] it is shown that the one-letter languages in MAT are regular.

3 Using variable thickness to simulate priority

It is known (see, for ex., [6]) that Rewriting P systems which made use of priority on the evolution rules are able to generate every Recursively Enumerable language. Neverless, the priority relation among evolution rules is a feature of a formal language inspiration, which does not seem a “realistic” one.

In the next proof, we prove that if we use the feature that modifies the thickness of a membrane, we are able to generate every RE language with RP systems that do not make use of priority. In fact, by modifying the thickness of the membrane we are able to simulate, in the correct order, the productions of a generic matrix grammar with appearance checking.

With $RP(nPri, \delta, \tau)$, we denote the family of languages generated by Rewriting P systems without priority on the evolution rules and which made use of variable thickness (i.e. both δ and τ operations).

Theorem 1 $RP(nPri, \delta, \tau) = RE$

Proof The inclusion $RP(nPri, \delta, \tau) \subseteq RE$ follows directly from the Church-Turing thesis. We prove here the opposite inclusion. Consider a matrix grammar with appearance checking $G = (N, T, S, P, F)$ in binary normal form. We assume the matrices of the types 2, 3 and 4 labeled in a one-to-one manner; we label with $m_1, ..., m_{k'}$ the matrices of type 2, with $m_{k'+1}, \ldots, m_{k''}$ the matrices of type 3 and with $m_{k''+1}, \ldots, m_k$ the matrices of type 4 $(0 \leq k' < k, 1 \leq k'' \leq k)$.

We show how to construct a Rewriting P System (of degree k + 2) without priority but with variable thickness that generates the same language of G:

$$\Pi \quad = \quad (V, \mu, M_1, M_2, \ldots, M_{k+1}, M_{k+2}, R_1, R_2, \ldots, R_{k+1}, R_{k+2}, M_{k+2})$$

where

- $V = N_1 \cup N_2 \cup \{E, \dagger, F, F', F_2, F_3, F_4\} \cup T \cup \{X', X_2, X_3 | X \in N_1\}$
- $\mu = [_0[_{k+1}[_1]_1[_2]_2] \cdots [_k]_k]_{k+1}]_0$
- $M_{k+1} = \{XAE | S \rightarrow XA\, rule\, of\, the\, matrix\, of\, type\, 1\} \cup \{F\}$
- $M_0 = \emptyset$
- $M_h = \dagger, for\, 1 \leq h \leq k$
- $R_i(1 \leq i \leq k') = \{A \rightarrow x\delta(out) | m_i : (X \rightarrow Y, A \rightarrow x)\, type\, 2\, matrix,$
  $x \in (N_2 \cup T)^*\} \cup \{F' \rightarrow F\tau(out)\}$
- $R_j(k' < j \leq k'') = \{Y' \rightarrow Y_2\tau\} \cup \{F' \rightarrow F_2\tau\} \cup \{Y_2 \rightarrow Y_3\} \cup \{F_2 \rightarrow F_3\tau\} \cup$
  $\cup \{A \rightarrow \dagger | m_j : (X \rightarrow Y, A \rightarrow \dagger)\, type\, 3\, matrix\} \cup \{F_3 \rightarrow F_4\delta\} \cup$
  $\cup \{Y_3 \rightarrow Y(out)\} \cup \{F_4 \rightarrow F(out)\}$
- $R_h(k'' < h \leq k) = \{A \rightarrow x_2\tau(out) | m_h : (X \rightarrow x_1, A \rightarrow x_2)\, type\, 4\, matrix,$
  $x_1, x_2 \in T^*\} \cup \{F' \rightarrow F\tau(out)\}$
- $R_{k+1} = \{X \rightarrow Y(in_w) | m_w : (X \rightarrow Y, A \rightarrow x)\, type\, 2\, matrix, x \in (N_2 \cup T)^*\} \cup$
  $\cup \{X \rightarrow Y'(in_w) | m_w : (X \rightarrow Y, A \rightarrow \dagger)\, type\, 3\, matrix\} \cup$
  $\cup \{X \rightarrow x_1(in_w) | m_w : (X \rightarrow x_1, A \rightarrow x_2)\, type\, 4\, matrix, x_1, x_2 \in T^*\} \cup$
  $\cup \{E \rightarrow \lambda(out)\} \cup \{F \rightarrow \lambda\} \cup \{F \rightarrow F'(in_r), 1 \leq r \leq k\} \cup \{\dagger \rightarrow \dagger\}$
- $R_0 = \{\alpha \rightarrow \alpha | \alpha \in V - T\}$

The most external membrane (M_0) is the output one. This membrane contains a membrane (M_{k+1}) used to control the simulation of the matrices. Inside this membrane there are k membranes, one for every matrix of the matrix grammar we have to simulate.

Consider the strings XwE (initially we have XAE) and F in membrane M_{k+1} with $w \in (N_2 \cup T)^*$. On the string F we can apply the following rules:

- $F \to \lambda$. This eliminates the string F.
- $F \to F^{'}(in_r)$, with $1 \leq r \leq k$. These rules send the string $F^{'}$ in a membrane corresponding to a matrix of the matrix grammar system.

On the string XwE we can apply one of the following rules:

- $E \to \lambda(out)$. This rule eliminates the symbol E from the string and it sends the obtained string in membrane 0, the output one.
- $X \to Y(in_w)$, where w is a label of a membrane associated with a type 2 matrix. These rules simulate the first production of a type 2 matrix and they send the string to the corresponding membrane.
- $X \to Y^{'}(in_w)$, where w is a label of a membrane associated with a type 3 matrix. These rules simulate the first production of a type 3 matrix and they send the string to the corresponding membrane.
- $X \to x_1(in_w)$, where w is a label of a membrane associated with a type 4 matrix. These rules simulate the first production of a type 4 matrix and they send the string to the corresponding membrane.

Consider what happens if we simulate the first production of a matrix on the string XwE and we send the obtained string in the corresponding membrane i ($1 \leq i \leq k$), and, at the same time, a rule $F \to F^{'}(in_j)$ with $j \neq i$ or a rule $F \to \lambda$ on F.

The string obtained from XwE is sent ALONE in membrane i.

If the membrane i is a membrane that simulates a type 2 matrix, we send in that membrane a string of the form YwE. If this string contains the corresponding symbol $A \in N_2$ we have to apply the rule $A \to x\delta(out)$. The membrane is dissolved, thus the string $\dagger$, present in the membrane, reaches the membrane $k+1$ where we have the rule $\dagger \to \dagger$: the computation will never stops. If the string YwE doesn't contain a symbol $A \in N_2$, the string cannot further evolve and, as we will see, no string can reach the output membrane.

If the membrane i is used to simulate a type 4 matrix, the situation is similar. The only difference is in the string we send: it is of the form x_1wE, where $x_1 \in T^*$.

Finally, if i corresponds to a type 3 matrix, we send in the membrane a string of the form $Y^{'}wE$. We can either apply a rule $Y^{'} \to Y_2\delta$, which dissolves the membrane and, consequently, sends the string $\dagger$ in membrane $k+1$, or, if the string contains the corresponding symbol $A \in N_2$, we can apply the rule $A \to \dagger$ that introduces the trap symbol $\dagger$ into the string. In both cases, the computation will never halts or no string will reach the output membrane.

The only way to correctly simulate a matrix is to apply, at the same time, the rules that send in the same membrane the strings obtained from XwE and F.

To simulate a type 2 matrix we have to apply a rule $X \rightarrow Y(in_i)$ on XwE and a rule $F \rightarrow F^{'}(in_i)$ on F. In this way, we get the strings YwE and $F^{'}$. Then these strings are sent to membrane i.

In membrane i, we have to apply the rule $F^{'} \rightarrow F\tau$ on $F^{'}$ and the rule $A \rightarrow x\delta(out)$ on YwE; in this way we send back in membrane $k+1$ the strings F and Yw_1xw_2E (with $w_1Aw_2 = w$). The thickness of the membrane remains unchanged (one rule uses the symbol δ while the other rule uses the symbol τ). Thus, we have correctly simulated the productions of a type 2 matrix on XwE (and the thickness of membrane i is still 1); we can proceed with the simulation of another matrix.

Note that if the symbol A is not present in XwE, this string can never leave the membrane, thus it cannot reach the output membrane.

To simulate a type 4 matrix, the process is quite similar. The only difference is that we have to apply a rule $X \rightarrow x_1(in_i)$ on XwE, where i is a label of a membrane used to simulate a type 4 matrix and x_1 is a terminal string.

The process of simulating a type 3 matrix is quite different. To simulate the productions of such a matrix we have to apply a rule $X \rightarrow Y^{'}(in_i)$ on XwE and the rule $F \rightarrow F^{'}(in_i)$ on F. We get the strings $Y^{'}wE$ and $F^{'}$ and these strings are sent to membrane i. Here, we have a 4 step process:

Step 1 (we control if the string $Y^{'}wE$ is arrived alone).

On $Y^{'}wE$ we can apply either the rule $Y^{'} \rightarrow Y_2\delta$ or the rule $A \rightarrow \dagger$ (if $Y^{'}wE$ contains the symbol A). At the same time, on $F^{'}$ we have to apply the rule $F^{'} \rightarrow F_2\tau$.

If we apply the rule $A \rightarrow \dagger$ on $Y^{'}wE$, we introduce the trap symbol $\dagger$ into the string; even if the string will reaches the output membrane, the computation will never halts. Otherwise, we get the strings Y_2wE and F_2. The thickness of the membrane remains unchanged (one rule uses the symbol δ while the other uses the symbol τ).

Step 2 (we increment the thickness of membrane).

On the string of the form Y_2wE we can apply either the rule $A \rightarrow \dagger$ or the rule $Y_2 \rightarrow Y_3$. At the same time, on F_2 we have to apply the rule $F_2 \rightarrow F_3\tau$.

If we apply the rule $A \rightarrow \dagger$ on Y_2wE, the considerations of the previous step are still valid. Otherwise, we get the strings Y_3wE and F_3. The thickness of the membrane is now 2, due to the symbol τ in the rule $F_2 \rightarrow F_3\tau$.

Step 3 (we execute the appearance checking on the string Y_3wE).

On the string of the form Y_3wE we can apply only the rule $A \rightarrow \dagger$ (if the string contains the symbol A). We cannot apply the rule $Y_3 \rightarrow Y\tau(out)$ because the thickness of the membrane is 2, thus the string cannot pass through the membrane. At the same time, on F_3 we have to apply the rule $F_3 \rightarrow F_4\delta$.

Consequently, if a symbol A is present in Y_3wE, <u>we have to apply</u> the rule $A \rightarrow \dagger$ that introduces the trap symbol. If A is not present, no rule can be applied on this string. The thickness of the membrane return to 1, due to the symbol δ in $F_3 \rightarrow F_4\delta$.

Step 4 (we conclude the simulation of a type 3 matrix).

The thickness of the membrane is now 1, so, on the string of the form Y_3wE, we can apply the rule $Y_3 \rightarrow Y\delta$. At the same time, on F_4 we have to apply the rule

$F_4 \rightarrow F\tau$.

The thickness of the membrane remains unchanged. We send back in membrane $k+1$ the string F and a string of the form YwE in which we have correctly simulated a type 3 matrix. We can start the simulation of another matrix.

We conclude this proof by illustrating the use of the rules $E \rightarrow \lambda(out)$ and $F \rightarrow \lambda$.

If we apply the rule $F \rightarrow \lambda$ on F and, at the same time, a rule that simulates the first production of a matrix on XwE, we delete the string F and we send the other string to a membrane labelled with a number between 1 and k. As we previously shown, if this second string is sent alone to a membrane with a label between 1 and k, the computation will never halts.

If we apply the rule $E \rightarrow \lambda(out)$ on a string of the form XwE, we send a string of the form Xw to the output membrane. The computation will never stops due to the rule $X \rightarrow X$ in that membrane. The same is true if we apply such a rule to a string of the form wE in which w contains non terminal symbols.

Thus, consider a string of the form vE in which v is a terminal string (we can obtain such a string after the simulation of a type 4 matrix). If we apply the rule $E \rightarrow \lambda(out)$ on vE we send the terminal string v in the output membrane; no other rule is applied on this string.

In membrane $k+1$ we can apply on F one of the rules $F \rightarrow F^{'}(in_i)$ or the rule $F \rightarrow \lambda$. If we apply a rule $F \rightarrow F^{'}(in_i)$ we send in membrane i a string $F^{'}$. On this string we have to apply the rule $F^{'} \rightarrow F\tau(out)$ (if i is a label of a membrane that simulates a type 2 or type 4 matrix) or the rules $F^{'} \rightarrow F_2\tau, F_2 \rightarrow F_3\tau, F_3 \rightarrow F_4\delta, F_4 \rightarrow F\tau(out)$ (if i is a label of a membrane that simulates a type 3 matrix). As one can see, in both cases we send back in membrane $k+1$ the string F and the thickness of the membrane i becomes 2.

On F we can apply now the rule $F \rightarrow \lambda$ or one of the rules $F \rightarrow F^{'}(in_j)$, with $j \neq i$, because the membrane i has thickness 2. If we apply one of the rules of the second type, we get the same string F in membrane $k+1$ and another membrane becomes of thickness 2.

It's easy to see that, after a while, all the membranes with a label between 1 and k become of thickness 2. So, if we still do not have applied the rule $F \rightarrow \lambda$, we have to apply it. The string F disappear and the computation stops.

In the output membrane we get exactly the strings of terminal symbols generated by the grammar G, that is $L(G) = L(\pi)$. $\diamondsuit$

Informally, the key concept in the previous proof is the collaboration between the string XwE and the string F. The computation can correctly terminate only when the two strings follow the same path between the membranes structure. If both strings use the same membrane at the same time, the computation can correctly proceed. Otherwise a membrane will be dissolved and a string $\dagger$ will reach the control membrane where the computation will proceed forever. For the same reason, the deletion of the string F cannot be done before sending the other string to the output membrane.

The difficult in simulating the production of a matrix grammar with appearance checking lies in the type 3 matrix: we have to apply the productions that introduce the trap symbol only if there are certain symbols in the string; otherwise these productions do not have to be applied. This can be easily controlled using priorities, but it seems difficult to get the same result without priorities. Nevertheless, by modifying the thickness of the membranes, we are able to simulate priority, as illustrated in the proof.

4 Using electrical charges to simplify membrane structure

Another feature in membrane systems which does not seem of a natural inspiration is relative to the communication of objects through membranes: the rules have to specify the label of the membrane where the objects have to be sent after the application of the rule itself. A less restrictive feature consider, as previously described, electrical charges associated with both the objects and the membranes.

We show now that this feature, introduced with the goal of obtaining more realistic systems, can be useful under different aspects. For example, it allows us to simplify the structure of the membrane systems: in fact, the communication of the strings can be controlled with few general rules, because we do not have to specify the precise label of the membrane where the strings have to go (as in the models presented in [6]), but only the subset of membranes we are interested in: positive or negative ones (of course, we have to control that the objects do not reach "wrong" membranes). We can, in this way, define P systems with a limited number of rules per membrane and with a membrane structure of limited depth.

We illustrate this in the following, using Rewriting P system with priority. With $RP^{\pm}(Pri,\, n\delta)$ we denote the family of languages generated by Rewriting P systems with electrical charges which use priority on evolution rules and which do not use the operation which allows to dissolve the membranes.

Definition A RP system is in double_2_ normal_form if it is of depth 2 and in each membrane we have 2 rewriting rules.

Theorem 2 Every RE language can be generated by a P system $RP^{\pm}(Pri,\, n\delta)$ in double_2_normal_form.

Proof Consider a matrix grammar with appearance checking $G = (N, T, S, P, F)$ in the normal form previously described. We assume the matrices labeled in a one-to-one manner. With $m_1, \ldots, m_{k_1}$ we label the matrices of type 2, with $m_{k_1+1}, \ldots, m_{k_2}$ we label the matrices of type 3 and with $m_{k_3+1}, \ldots, m_k$ we label the matrices of type 4. Moreover, we label the symbols in N with $N_1, \ldots, N_h$.

We show how to construct a Rewriting Super Cell system of depth 2 with 2 rewriting rules in each membrane that generates the same language of G:

$$\Pi \;=\; (V, \mu, M_0, M_1, \ldots M_k, \ldots, M_{k+h}, M_{k+h+1}, (R_1, \rho_1), \ldots, (R_{k+h+1}, \rho_{k+h+1}), 0)$$

where

- $V = N_1 \cup N_2 \cup \{Z, Z^{'}, Z^{''}\} \cup \{C_i | 1 \leq i \leq h\} \cup T,$
- $\mu = [_0 [_1]_1^+ \cdots [_k]_k^+ [_{k+1}]_{k+1}^- \cdots [_{k+h+1}]_{k+h+1}^-]_0$

- $M_0 = \{Z^{'}ZXA | S \rightarrow XA$ is the rule of a matrix of type 1$\}$
- $M_1, ..., M_{k+h+1}$ are empty
- $R_0 = \{r_{0,1} : z \rightarrow \lambda(-)\} \cup \{r_{0,2} : Z^{'} \rightarrow Z^{'}(-)\}$
- $R_\alpha = \{r_{\alpha,1} : X \rightarrow ZY\} \cup \{r_{\alpha,2} : A \rightarrow X(out)\}$, $1 \leq \alpha \leq k_1$, (type 2 matrices)
- $R_\beta = \{r_{\beta,1} : A \rightarrow A\} \cup \{r_{\beta,2} : X \rightarrow ZY(out)\}$, $k_1 + 1 \leq \beta \leq k_2$, (type 3 matrices)
- $R_\gamma = \{r_{\gamma,1} : X \rightarrow x_1C_1\} \cup \{r_{\gamma,2} : A \rightarrow x(out)\}$, $k_2 + 1 \leq \gamma \leq k$, (type 4 matrices)
- $R_\psi = \{r_{\psi,1} : N_i \rightarrow N_i\} \cup \{r_{\psi,2} : C_i \rightarrow C_{i+1}(out)\}, k + 1 \leq \psi \leq k + h,$
  $1 \leq i \leq h, \psi = i + k$
- $R_{h+k+1} = \{r_{h+k+1,1} : Z^{'} \rightarrow \lambda\} \cup \{r_{h+k+1,2} : Z^{''} \rightarrow \lambda(out)\}$
- $\rho_i : r_{i,1} > r_{i,2},$ for every i, $0 \leq i \leq h + k + 1$

In other words, we place into the skin membrane several membranes with positive charge, one for each matrix in G; each membrane simulates the productions of a matrix in G. Moreover, we place into the skin membrane several membranes with negative charge, one for every nonterminal symbol in G plus one used to stop the computation; these membranes are used to be sure that the generated strings do not contain non terminal symbols.

Consider the string $Z^{'}ZHw$ in membrane 0 with $w \in (N_2 \cup T)^*$ and $H \in N_1$ (initially we have $Z^{'}ZXA$). We have to apply the production $Z \rightarrow \lambda(-)$ and we get the string $Z^{'}Xw$. This string is sent to a membrane of positive charge, in which we simulate the productions of a type 2, 3 or 4 matrix.

Membrane simulating a type 2 matrix: If the string is sent to a membrane corresponding to a type 2 matrix, we have to apply a rule of the form $X \rightarrow ZY$ (which simulates the first production of the type 2 matrix) and a rule of the form $A \rightarrow x(out)$ (which simulates the second production of a type 2 matrix). The first production of a matrix of type 2 in the binary normal form cannot be of the form $X \rightarrow X$, as one can see from the description of the normal form in [2]. Thus, if $H \neq X$ (i.e. the string contains the symbol X) we have to apply this rule and we can do it only one time (the string cannot evolve forever in this membrane due to a rule $X \rightarrow X$). Otherwise, we cannot apply this rule, and the symbol Z is not reinserted in the string. Either if we have applied the previous rule or if $H \neq X$, we have to apply the rule $A \rightarrow x(out)$. If the symbol A is not in w, the string cannot further evolve and it will not reach the output membrane; otherwise the obtained string is sent back in the skin membrane. As said before, if the production $X \rightarrow ZY$ have not been applied, we have now a string of the form $Z^{'}Hw^{'}$, i.e. a string without the symbol Z in it. Thus, in the skin membrane we have to apply the rule $Z^{'} \rightarrow Z^{'}(+)$, which sends the string in a membrane with negative charge. It easy to see that there is no membrane that sends back the string in the skin membrane (the string does not contains the symbol $Z^{''}$ nor a symbol C_i). The computation can correctly proceed only if the membrane correctly simulates the corresponding type 2 matrix. In fact, if we apply the production $X \rightarrow ZY$ and if the symbol A is in w, we can apply the rule $A \rightarrow x(out)$, that sends back in membrane 0 the string $Z^{'}ZYw^{'}$, that is ready to simulate another matrix.

Membrane simulating a type 3 matrix If the string $Z^{'}Hw$ is sent to a membrane of type 3, we have to apply the rules $A \rightarrow A$ and $X \rightarrow ZY$. We have the following possibilities: if the string contains the symbol A, we have to apply forever the production $A \rightarrow A$ (due to the priority), thus the computation will never stop and no string will be produced. If the symbol A is not in the string, we can apply the other production $X \rightarrow ZY(out)$. If $H \neq X$ the string cannot further evolve and it will remain in this membrane, otherwise we correctly simulate a type 3 matrix and the string is sent back in membrane 0 where we can start the simulation of another matrix.

Membrane simulating a type 4 matrix If the string $Z^{'}Hw$ is sent to a membrane of type 4, we have to apply the rules $X \rightarrow C_1x_1$ and $A \rightarrow x(out)$. If the string does not contain the symbol A, it will remain in the membrane forever. If the string contains the symbol A but it doesn't contain the symbol X, we can apply the rule $A \rightarrow x(out)$. The obtained string doesn't contain the symbol Z nor the symbol C_1. As we have seen before for the type 2 matrix, this string will reach a membrane with negative charge but it will never exit from that, thus no string will be generated in this way. The matrix will be correctly simulated only if $H = X$ and the string contains the symbol A. In this case, we first apply the rule $X \rightarrow C_1x_1$ and then the rule $A \rightarrow x(out)$. In the skin membrane we get a string of the form $Z^{'}C_1w$.

Thus, we are able to simulate the productions of every type of matrix in the correct order. When the string reaches a membrane that corresponds to a type 4 matrix, the phase of simulating the production of the matrices has to be ended, and we have to control that the obtained string does not contain non terminal symbols. This is done with the negative charged membranes. Consider a string of the form $Z^{'}C_1w$ in membrane 0. Obviously, the production $Z \rightarrow \lambda$ cannot be applied, because the string does not contain the symbol Z. Thus, we have to apply the rule $Z^{'} \rightarrow Z^{'}(+)$. The obtained string $Z^{'}C_1w$ is sent to a membrane with negative charge. There is only one membrane with a production that involved C_1: the membrane used to control the presence of the non terminal N_1, that is the membrane k+1; it contains the productions $N_1 \rightarrow N_1$ and $C_1 \rightarrow C_2(out)$. If the string reaches a different membrane, it cannot further evolve and no string reaches the output membrane. Otherwise, we can test the presence of the non terminal N_1 in the string: if N_1 is in the string, we have to apply forever the production $N_1 \rightarrow N_1$, otherwise we can apply the production $C_1 \rightarrow C_2(out)$ and we send back in membrane 0 the string $Z^{'}C_2w$. Here we can apply again the rule $Z^{'} \rightarrow Z^{'}(+)$ to send the string in a membrane with negative charge.

Now, the "correct" one is the membrane that tests the presence of the non terminal symbol N_2. If the string reaches another membrane, it will be blocked. If it reaches the membrane $k+2$ and the string contains the symbol N_2 the computation will never halt. Otherwise we get in membrane 0 the string $Z^{'}C_3w$. The computation proceeds in this way until we test all non terminal symbols, using the sequence

$C_1, C_2, \ldots, C_h$. In the membrane that controls the presence of the h-th non terminal symbol (the last one) we have the production $C_h \to Z''(out)$ (Z'' tell us that we have checked the presence of all non terminal symbols). The string is sent back in membrane 0 where we have to apply again the rule $Z' \to Z'(+)$. The string $Z'Z''w$ is sent to a membrane with negative charge. If it reaches membrane $k+h+1$ we apply the rules $Z' \to \lambda$ and $Z'' \to \lambda(out)$ that sends back in membrane 0 (the output one) the terminal string w; otherwise the string will be blocked in one of the other membrane with negative charge.

Thus, in the output membrane we get exactly the strings of terminal symbols generated by G, that is $L(G) = L(\Pi)$. $\diamondsuit$

Informally, the communication of the strings (from the skin membrane to the other membranes) is accomplished using two general rules, because we do not have to specify the label of the membrane where we send the string: with one rule we start the simulation of a matrix, while with the other rule we stop the simulation and we start the phase in which we control if the string is a terminal one. We can control if a string reaches a "wrong" membrane using the rules in the same membrane, as we've shown in the proof.

Note that the proof presented here does not show how to build a system in double_2_normal_form starting from a generic RP systems with priority (in a direct way). Of course, given a RP system, we can build an equivalent type 0 grammar, from this we can build the equivalent matrix grammar with appearance checking and finally we can obtain an equivalent RP system in the normal form.

5 Using membrane division to solve NP complete problems in polynomial time with P systems

Could we use P systems to solve complex problems in an efficient way? In [8] one considers a variant of P systems in which a membrane can be divided in two membranes; each new membrane contains the same objects and rules of the starting membrane. In [4] one considers the division of a membrane in an arbitrary finite number of membranes. In [8] and [4] it is shown how to solve in linear time (with respect to the input length) two well known NP Complete problems, Satisfiability and Hamiltonian Path, using P systems with active membranes. The systems in these papers use division for elementary membranes and for non elementary membranes (i.e. membranes with one or more membranes inside).

We show now that division for elementary membranes suffice to build P systems able to solve complex problems in an efficient way. In particular, we show how to build a P system, with division for elementary membranes only (called P Systems with Elementary Active Membranes), able to solve the SAT problem in linear time.

Theorem 3: The SAT problem can be solved in linear time (with respect to the number of variables and the number of clauses) by a P system with elementary active membranes.

Proof: Consider a boolean expression in conjunctive normal form

$$\alpha = C_1 \wedge C_2 \wedge ... \wedge C_m$$

for some $m \geq 1$, with

$$C_i = y_{i,1} \vee y_{i,2} \vee ... \vee y_{i,p_i}$$

where $p_i \geq 1$, and $y_{i,j} \in \{x_k, \neg x_k | 1 \leq k \leq n\}$, for each $1 \leq i \leq m$, $1 \leq j \leq p_i$.

We build the P system

$$\Pi = (V, T, H, \mu, \omega_0, \omega_1, R)$$

where

- $V = \{a_i, t_i, f_i | 1 \leq i \leq n\} \cup \{r_i | 1 \leq i \leq m\} \cup \{W_i | 1 \leq i \leq m+1\} \cup$
  $\cup \{t\} \cup \{Z_i | 0 \leq i \leq n\}$
- $T = \{t\}$
- $H = \{0, 1\}$
- $\mu = [_0 [_1]_1^0]_0^0$
- $\omega_0 = \lambda$
- $\omega_1 = a_1 a_2 ... a_n Z_0$

while the set R contains the following rules:
1. $[a_i]_1^0 \to [t_i]_1^0 [f_i]_1^0$, $1 \leq i \leq n$
We substitute one variable a_i in membrane 1 with two variables t_i and f_i. The membrane is divided in two membranes (the charge remains neutral for both membranes). In n steps we get all 2^n truth assignments for the n variables; each truth assignment is in a membrane labelled with 1.
2. $[Z_k \to Z_{k+1}]_1^0$, $0 \leq k \leq n-2$
3. $[Z_{n-1} \to Z_n W_1]_1^0$
4. $[Z_n]_1^0 \to []_1^+ \lambda$
We count n steps, the time needed to produce all the truth assignments. The step $n-1$ introduces the symbol W_1 used in the next steps, while the step n changes the charge of the membranes labelled with 1.
5. $[t_i \to r_{h_1} ... r_{h_j}]_1^+$, $1 \leq i \leq n$, $1 \leq h_j \leq m$ and x_i is in the clauses h_1, ..., h_j
6. $[f_i \to r_{h_1} ... r_{h_j}]_1^+$, $1 \leq i \leq n$, $1 \leq h_j \leq m$ and $\neg x_i$ is in the clauses h_1, ..., h_j
In one step, each symbol t_i is replaced with some symbols r_{h_j}, indicating the clauses satisfied if we set $x_i = true$; each symbol f_i is replaced with some symbols, indicating the clauses satisfied if we set $x_i = false$ (i.e. $\neg x_i = true$).

After this step, we start the "VERIFY STEPS": we have to verify if there is at least one membrane, labelled with 1, in which we get all symbols $r_1, r_2, ..., r_m$ (at least one symbol r_i for every i). In fact, this means that there is a truth assignment satisfying all the clauses.
7. $[r_1]_1^+ \to []_1^- r_1$
We first verify the presence of the symbol r_1 in membranes labelled with 1. Every membrane containing r_1 (i.e. every membrane with a truth assignment satisfying the first clause) sends this symbol outside (in membrane 0) and changes its charge

from $+$ to $-$. The membranes not containing this symbol cannot further proceed their computation.
8. $r_1[\,]_1^- \rightarrow [r_0]_1^+$
9. $[r_i \rightarrow r_{i-1}]_1^-$, $1 \leq i \leq m$
10. $[W_i \rightarrow W_{i+1}]_1^-$, $1 \leq i \leq m$
The membranes with negative charge can continue the computation. They increase the index of the symbols W_i (the counters the satisfied clauses) and decrease the index of the symbols r_i. The symbols r_1 in membrane 0 are sent back (as r_0) to the negative charged membranes labelled with 1, to change their charge from $-$ to $+$.

After applying the rules of type 8, 9, and 10 (in parallel) we can re-apply the rules of type 7 followed again by the application of the rules of type 8, 9 and 10. In $2m$ steps we verify the existence of a membrane containing all symbols r_i. It's easy to see that if a membrane does not contain a symbol r_i, the computation in that membrane halts in less than $2m$ steps. As a consequence, that membrane will not contain the symbol W_{m+1}. The membranes containing this symbol are the membranes containing all symbols r_i when the VERIFY STEPS started, i.e. the membranes with a truth assignment satisfying all the clauses.
11. $[W_{m+1}]_1^+ \rightarrow [\,]_1^+ t$
If a membrane labelled with 1 executes all $2m$ verify steps, it contains the symbol W_{m+1}. Thus we send out to membrane 0 the symbol t, indicating that there is a truth assignment satisfying all the clauses.
12. $[t]_0^0 \rightarrow [\,]_0^+ t$
A symbol t is sent outside membrane 0 and the charge of this membrane is changed from 0 to $+$. No further computation is possible. Thus, we have to look at the output of membrane 0 after $n + 2m + 5$ steps. If we get the symbol t, it means that there is a truth assignment satisfying α; otherwise, the formula is not satisfiable. •

6 Conclusions

In papers like [7], [8] and [9], is pointed out the importance of consider variants of P systems which use features of biochemical inspiration, to obtain more "realistic" systems which can be implemented either in biochemical media or in electronic media.

We've presented here some results, to underline the fact that a feature of biochemical inspiration can be useful not only to replace other features of different inspiration (for example, of a formal language inspiration, like priority over evolution rules). These features can be used to obtain simpler and faster models too. We've illustrated this, by showing that variable thickness can be used to replace priority, while electrical charges can be used to obtain systems with a simple membrane structure (systems of depth two with two rules per membrane). Moreover, we've shown how to build P systems able to solve the Satisfiability problem (a well known NP complete problem) in linear time, using division for elementary membranes only.

References

[1] J. Dassow, Gh. Paun, On the power of membrane computing, J. Univ. Computer Sci., 5, 2 (1999), 33-49.

[2] J. Dassow, Gh. Paun, Regulated Rewriting in Formal Language Theory, Springer-Verlag, Berlin, 1989.

[3] D. Hauschildt, M. Jantzen, Petri nets algorithms in the theory of matrix grammars, Acta Informatica, 31 (1994), 719-728.

[4] S. N. Krishna, R. Rama, A variant of P systems with active membranes: Solving NP-complete problems, Romanian J. of Information Science and Technology, 2, 4 (1999).

[5] Gh Paun, Computing with membranes. An introduction, Bulletin of the EATCS, 67 (Febr. 1999), 139-152.

[6] Gh. Paun, Computing with membranes, submitted, 1998 (see also TUCS Research Report No 208, November 1998 http://www.tucs.fi).

[7] Gh. Paun, Computing with membranes - a variant: P systems with polarized membranes, Auckland Univ., CDMTCS Report No. 098, 1999.

[8] Gh. Paun, P systems with active membranes: attacking NP complete problems, submitted 1999 (see also CDMTCS Research report No. 102, 1999, Auckland Univ., New Zeland, www.cs.auckland.ac.nz/CDMTCS)

[9] Gh. Paun, G.. Rozenberg, A. Salomaa, Membrane computing with external output, submitted, 1999 (see also TUCS Research Report No. 218, December 1998, http://www.tucs.fi).

[10] Gh. Paun, S. Yu, On synchronization in P systems, submitted, 1999 (see also CS Department TR No 539, Univ. of Western Ontario, London, Ontario, 1999, www.csd.uwo.ca/faculty/syu/TR539.html).

[11] Gh. Paun, T. Yokomori, Membrane computing based on splicing, proc. of 5th DIMACS Workshop on DNA Based Computers, 1999, 213-227.

[12] I. Petre, A normal form for P systems, Bulletin of EATCS, 67 (Febr. 1999), 165-172.

[13] G. Rozenberg, A. Salomaa, eds. , Handbook of Formal Languages, Springer-Verlag, Heidelberg, 1997.