Lecture Notes in Computer Science 2055

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Maurice Margenstern Yurii Rogozhin (Eds.)

Machines, Computations, and Universality

Third International Conference, MCU 2001
Chişinău, Moldova, May 23-27, 2001
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Maurice Margenstern
Université de Metz
LITA, UFR MIM
Ile du Saulcy, 57045 Metz Cedex, France
E-mail: margens@lita.univ-metz.fr

Yurii Rogozhin
Institute of Mathematics and Computer Science
of the Academy of Sciences of Moldova
5 Academiei str, 2028 Chişinău, Moldova

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Machines, computations, and universality : third international conference / MCU 2001, Chisinau, Moldova, May 23 - 27, 2001. Maurice Margenstern ; Yurii Rogozhin (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2055)
ISBN 3-540-42121-1

CR Subject Classification (1998): F.1, F.4, F.3, F.2

ISSN 0302-9743
ISBN 3-540-42121-1 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Steingräber Satztechnik GmbH
Printed on acid-free paper SPIN 10781543 06/3142 5 4 3 2 1 0

Preface

In the first part of the present volume of *LNCS*, the reader will find the invited talks given at the MCU 2001 conference. In the second part, he/she will find the contributions that were presented at the conference after selection. In both cases, papers are arranged in the alphabetical order of the authors.

MCU 2001 is the third conference in theoretical computer science, *Machines, computations and universality*, formerly, *Machines et calculs universels*. Both previous conferences, MCU'95 and MCU'98, were organized by Maurice Margenstern in Paris and in Metz (France), respectively.

From the very beginning, MCU conferences have been an international scientific event. For the third conference, in order to stress that aspect, it was decided to hold it outside France. Moldova was chosen thanks to the close cooperation between the present chairmen of MCU 2001.

MCU 2001 also aims at high scientific standards. We hope that the present volume will convince the reader that the tradition of previous conferences have been upheld by this one. Cellular automata and molecular computing are well represented in this volume. And this is also the case for quantum computing, formal languages, and the theory of automata. MCU 2001 does not fail its tradition of providing our community with important results on Turing machines.

We take this opportunity to thank the referees of the submitted papers for their very efficient work. The members of the program committee gave us decisive help on this occasion. Thanks to them, namely, Erzsebet Csuhaj-Varjú, Gabriel Ciobanu, Serge Grigorieff, Manfred Kudlek, Yuri Matiyasevich, Liudmila Pavlotskaya, Arto Salomaa, and Mephodii Ratsa, we can offer the reader this volume of *LNCS*.

The local organizing committee includes E.Boian, C.Ciubotaru, S.Cojocaru, A.Colesnicov, V.Demidova, G.Magariu, N.Magariu, L.Malahova, T.Tofan, T.Verlan, and S.Verlan. We would like to thank all these people for their work, especially Serghei Verlan for the preparation of this volume. We express our gratitude to all colleagues from the Institute of Mathematics and Computer Science of the Academy of Sciences of Moldova who have supported us in organizing this event. We have a special debt towards the director of the institute, Constantin Gaindric.

Special thanks are due to the town of Chişinău for its assistance, in particular to its mayor, Serafim Urecheanu.

MCU 2001 could not have been held without the help of significant supporters. We therefore thank the *Laboratoire d'Informatique Théorique et Appliquée*, *LITA*, the University of Metz, the *Pôle Universitaire Européen de Nancy-Metz*, and especially *INTAS* project 97-1259, namely, its western partners, the University of Metz and the University of Hamburg.

March 2001

Maurice Margenstern
Yurii Rogozhin

Organization

MCU 2001 was organized by the Laboratoire d'Informatique Théorique et Appliquée (LITA), University of Metz, Metz, France and the Institute of Mathematics and Computer Science, Academy of Sciences of Moldova, Chişinău, Moldova.

Program Commitee

Erzsebet Csuhaj-Varju	Hungarian Academy of Sciences, Hungary
Gabriel Ciobanu	A.I.Cuza Universty, Iaşi, România
Serge Grigorieff	University of Paris 7, France
Manfred Kudlek	University of Hamburg, Germany
Maurice Margenstern	**Co-Chair**, LITA, University of Metz, I.U.T. of Metz, France
Yuri Matiyasevich	Euler Institute, Steklov Institute, St. Petersburg, Russia
Liudmila Pavlotskaya	Moscow Power Engineering Institute, Russia
Yurii Rogozhin	**Co-Chair**, Institute of Mathematics and Computer Science, Chişinău, Moldova
Mephodii Ratsa	Institute of Mathematics and Computer Science, Chişinău, Moldova
Arto Salomaa	Academy of Finland and Turku Centre for Computer Science, Finland

Table of Contents

Invited Lectures

Three Small Universal Turing Machines 1
Claudio Baiocchi

Computation in Gene Networks 11
Asa Ben-Hur, Hava T. Siegelmann

Power, Puzzles and Properties of Entanglement 25
Jozef Gruska, Hiroshi Imai

Combinatorial and Computational Problems on Finite Sets of Words 69
Juhani Karhumäki

Computing with Membranes (P Systems): Universality Results 82
Carlos Martín-Vide, Gheorghe Păun

A Simple Universal Logic Element
and Cellular Automata for Reversible Computing 102
Kenichi Morita

Some Applications of the Decidability of DPDA's Equivalence. 114
Géraud Sénizergues

The Equivalence Problem for Computational Models:
Decidable and Undecidable Cases 133
Vladimir A. Zakharov

Two Normal Forms for Rewriting P Systems 153
Claudio Zandron, Claudio Ferretti, Giancarlo Mauri

Technical Contributions

On a Conjecture of Kůrka.
A Turing Machine with No Periodic Configurations 165
Vincent D. Blondel, Julien Cassaigne, Codrin Nichitiu

On the Transition Graphs of Turing Machines 177
Didier Caucal

JC-Nets 190
Gabriel Ciobanu, Mihai Rotaru

Nonterminal Complexity of Programmed Grammars 202
Henning Fernau

On the Number of Non-terminal Symbols
in Graph-Controlled, Programmed and Matrix Grammars 214
Rudolf Freund, Gheorghe Păun

A Direct Construction of a Universal Extended H System 226
Pierluigi Frisco

Speeding-Up Cellular Automata by Alternations 240
Chuzo Iwamoto, Katsuyuki Tateishi, Kenichi Morita, Katsunobu Imai

Efficient Universal Pushdown Cellular Automata
and Their Application to Complexity 252
Martin Kutrib

Firing Squad Synchronization Problem
on Bidimensional Cellular Automata with Communication Constraints 264
Salvatore La Torre, Margherita Napoli, Mimmo Parente

P Systems with Membrane Creation: Universality and Efficiency 276
Madhu Mutyam, Kamala Krithivasan

On the Computational Power of a Continuous-Space Optical Model
of Computation 288
Thomas J. Naughton, Damien Woods

On a P-optimal Proof System
for the Set of All Satisfiable Boolean Formulas (SAT) 300
Zenon Sadowski

D0L System + Watson-Crick Complementarity=Universal Computation . . 308
Petr Sosík

Author Index 321

Three Small Universal Turing Machines

Claudio Baiocchi

Dipartimento di Matematica, Università "La Sapienza" di Roma (Italy)
baiocchi@mat.uniroma1.it

Abstract. We are interested by "small" Universal Turing Machines (in short: UTMs), in the framework of $2, 3$ or 4 tape–symbols. In particular:

- 2 tape–symbols. Apart from the old 24–states machine constructed by Rogozhin in 1982, we know two recent examples requiring 22 states, one due to Rogozhin and one to the author.
- 3 tape–symbols. The best example we know, due to Rogozhin, requires 10 states. It uses a strategy quite hard to follow, in particular because even–length productions require a different treatment with respect to odd–length ones.
- 4 tape–symbols. The best known machines require 7 states. Among them, the Rogozhin's one require only 26 commands; the Robinson's one, though requiring 27 commands, fournishes an easier way to recover the output when the TM halts. In particular, Robinson asked for a 7×4 UTM with only 26 commands and an easy treatment of the output.

Here we will firstly construct a 7×4 UTM with an easy recover of the output which requires only 25 commands; then we will simulate such a machine by a (simple) 10×3 UTM and by a 19×2 UTM.

1 Tag Systems

Let us recall some notions needed later on. We are given an integer $d \geq 2$ (the *deletion number*); a "d–tag system" is nothing but a word–processor, acting on words W whose letters are the elements of a finite alphabet $\mathcal{A}$. For each letter $a \in \mathcal{A}$, a (possibly empty) word $p(a)$ is given, that is called the "production" of a; the letters whose production is empty are called *halting* letters. Words of length strictly less than d and words starting with a halting–letter are called halting–words. The action of the word–processor on the generic word W is specified by the following rules:

- if W is a halting–word, the word–processor stops;
- if not, W has the form $W = aUV$ where: a is a non–halting letter; U is a word of length $d-1$; V is another (possibly empty) word. The original word $W = aUV$ is then replaced with $W = Vp(a)$; and the work continues on the new W thus obtained.

For our purposes we can restrict ourselves to the case of deletion number $d = 2$; furthermore we can assume that all but one the halting letters are "virtual",

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 1–10, 2001.

because (due to the form of the initial word W) they will never appear as initial letters; and that the length of the word W to be processed will always remain strictly greater than 1 (in the following, when speaking of "tag systems", we will refer to such restricted case).

Following the general approach suggested by Minsky (see e.g. [Min]) we will now construct some TMs that simulate tag–systems; they will be *universal* because, in turn, TMs can be simulated by tag–systems[1].

2 Simulation of Tag–Systems

We will use the term "symbol" (or tape–symbol) to denote the elements of the TM's alphabet, and the term "string" to denote words constructed with tape–symbols; terms "letter" and "word" will instead refer to the alphabet $\mathcal{A}$ of the simulated tag–system. We will set $\mathcal{A} = \{a_0, a_1, a_2, \dots, a_n\}$, the unique actual halting letter being a_0; recall that the only case of halting corresponds to a word of length at least 2 whose first letter is a_0.

The initial tape of the simulating machine will be choosen in the form:

$$\mathcal{T} = \dots\dots \mathcal{P} \dots \mathcal{S} \dots\dots$$

where:

- dots denote zones of blanks;
- the string $\mathcal{S}$ encodes (in a form we will detail in a moment) the word W to be processed;
- the string $\mathcal{P}$ has the form:

$$\mathcal{P} = P_{n+1} P_n \cdots P_1 P_0 \tag{1}$$

  where P_j encodes, in a suitable form, the production $p(a_j)$ (if we do not specify the index of a letter $a \in \mathcal{A}$, we will denote by $P(a)$ the corresponding string in $\mathcal{P}$).

We will associate to any $a_j \in \mathcal{A}$ a positive integer N_j (again, if we do not specify the index of an $a \in \mathcal{A}$ we will write $N(a)$ instead of N_j). The definition of the function N will be given later on; for the moment let us say only that it must be injective. Once N has been choosen, we fix two tape–symbols μ, ν with $\mu \neq \nu$ and we set[2]:

$$\begin{cases} \text{if } W = b_1 b_2 \dots b_k \text{ is the initial word,} \\ \text{we choose } \mathcal{S} = \nu^{N(b_1)} \mu \nu^{N(b_2)} \mu \dots \mu \nu^{N(b_k)} \end{cases} \tag{2}$$

so that the exponent of ν will specify (the function N being injective) which letter we are dealing with.

[1] the details can be found e.g. in the self–contained and very clear paper [Rob]

[2] ν stay for "normal", μ for "marker"

In particular let W start with a non–halting letter a, whose production is $p(a) = a_{j_1} a_{j_2} \dots a_{j_p}$; in order to "elaborate" such a W we must append to the right end of $\mathcal{S}$ the string $\Sigma(a) = \mu\nu^{N_{j_1}} \mu\nu^{N_{j_2}} \mu \dots \nu^{N_{j_p}}$; and then kill the leftmost part of $\mathcal{S}$ (up to, and included, the second marker). In order to append $\Sigma(a)$ it could be useful to have, somewhere on the left of the scanned zone, a "mirror image" $\Sigma'(a)$ of $\Sigma(a)$; thus it will be sufficient to copy each symbol of $\Sigma'(a)$ at the end of $\mathcal{S}$[3].

The needed $\Sigma'(a)$ will in fact be the rightmost part of $P(a)$ (see formula (4) later on); the leftmost part of $P(a)$ being a "super–marker", signalling the end of $\Sigma'(a)$. Of course, before copying the needed string, we must "reach" it; in other words, by using the initial part $\nu^{N(a)}$ of $\mathcal{S}$, we must "neutralize" the part of $\mathcal{P}$ which is on the right of $P(a)$. This will be realized through a clever choice of the function N, which will be choosen in such a way that:

$$\left\{ \begin{array}{l} \text{if the initial letter of the word } W \text{ is } a_\ell, \\ \text{the corresponding initial string } \nu^{N_\ell} \text{ in } \mathcal{S} \text{ will force} \\ \text{the TM to "neutralize" } P_0, P_1, \dots P_{\ell-1}. \end{array} \right. \tag{3}$$

Such a neutralization will be the first part of the whole TM's job; the second part will copy the string; the third phase will recover the neutralized part of $\mathcal{P}$ and will annihilate the part of $\mathcal{S}$ corresponding to the two initial letters of the starting word W.

3 A 7×4 UTM with 25 Commands

We will choose as tape–symbols 0 (the blank) and α, β, γ. Concerning $\mathcal{S}$, we will choose α as the normal symbol and γ as the marker; concerning $\mathcal{P}$, the normal symbol will be 0 and the marker will be β; more precisely, a string like $\dots 00\beta$ will act as marker; a string like $\dots \beta 0 \beta$ will act as super–marker. We will fix the form of $\mathcal{P}$ by setting:

$$\left\{ \begin{array}{l} P_{n+1} = \beta;\ P(a) = 0\beta \text{ for any halting letter } a; \\ \text{for non halting letters, if } p(a) = b_1 b_2 \dots b_k, \text{ we choose:} \\ P(a) = 0\beta 0^{N(b_k)} 00 \beta 0^{N(b_{k-1})} 00 \beta \dots \beta 0^{N(b_2)} 00 \beta 0^{N(b_1)} 00 \beta. \end{array} \right. \tag{4}$$

Remark 1. In particular halting letters will all be encoded in $\mathcal{P}$ through the same string. This is not a problem, because virtual halting letters will never appear in first position in $\mathcal{S}$, thus their representations in $\mathcal{P}$ do not need to differ each one other (but their codes in $\mathcal{S}$ do need to differ...). On the other hand, if we sort $\mathcal{A}$ by putting all virtual halting letters at the end, we could suppress their representation in $\mathcal{P}$[4].

[3] $\Sigma'(a)$ is read from right to left, thus we ask for the mirror image. Of course we do not need a true copy: e.g. both markers and normal symbols used in $\mathcal{P}$ could differ from the ones (μ, ν) used in $\mathcal{S}$...

[4] but the leftmost part, $P_{n+1} = \beta$, is still needed

Remark 2. The neutralization of a P_j will simply consist in replacing each 0 by an α. Let us assume that the needed neutralization has been done; and let $P(a)$ be the rightmost non–neutralized part of $\mathcal{P}$. The string $P(a)$ ends with a substring like $\beta 0^x 00\beta$; its final part 00β will act as a first marker. Meeting such a marker, the TM will change it into $\alpha\alpha\beta$ and will append a marker γ on the right of $\mathcal{S}$. Coming back, the TM will encounter $\beta 0^x \alpha\alpha\beta$ and each 0 if any[5] will be changed into an α and that will force the TM to append an α. The work continues with the next (on the left) string like $\beta 0^y 00\beta$; and so on, until we reach (instead of a $\beta 0^z 00\beta$) the super–marker $\beta 0\beta$ (the leftmost β being the rightmost symbol of the closer P on the left, possibly P_{n+1}); this ends the second step of the work, and the restore phase will start.

Now we need an explicit formula for the function N; we will choose:

$$N_0 := 0; N_k := N_{k-1} + 1 + |p(a_k)| \qquad (k = 1, \ldots, n) \tag{5}$$

where $|X|$ denotes the length of the word X; remark that the "+1" ensures the injectivity of the function N.

Remark 3. In particular, see (2), the actual halting letter a_0 will be represented in $\mathcal{S}$ by an empty–string; its presence in the middle of W will correspond to a substring $\gamma\gamma$ in $\mathcal{S}$; the presence of a_0 in first (resp. last) position will correspond to the fact that the string $\mathcal{S}$ starts (resp. ends) with a γ. Halting–words in the tag–system thus correspond to $\mathcal{S}$ starting with a γ; in such a case the TM will simply halt.

With the choices (4) and (5) the reader will have no difficulties to check that property (3) holds if, in the first phase of the TM's work, each α of the initial part $\alpha^{N(a)}$ in $\mathcal{S}$ neutralizes up to and included the first substring 0β on its left; thus the initial TM's can be regulated by the following piece of table where, in first column, we indicate by arrows the direction the head is moving to:

Table 1. The first part of the TM's work.

Direction	State	0	α	β	γ	Description
$\rightarrow$	A	$\square$	$0\mathcal{L}B$	$\square$	Halt	The whole TM's work starts here
$\leftarrow$	B	$0\mathcal{L}B$	$0\mathcal{L}B$	$\beta\mathcal{L}C$	$\square$	Neutralization in $\mathcal{P}$
$\leftarrow$	C	$\alpha\mathcal{R}D$	$0\mathcal{L}B$	$\square$	$\square$	Doubt
$\rightarrow$	D	$\alpha\mathcal{R}D$	$0\mathcal{L}B$	$\beta\mathcal{R}D$	$\alpha\mathcal{L}E$	Neutralization in $\mathcal{S}$

[5] the letter a_0 corresponds to $x = 0$; see the following formula (5)

Remark 4. We want to point out that cells marked as $\boxed{\ }$ refer to cases we will never encounter if the initial tape corresponds to the simulation of a tag–system; in particular we could fill them with any command without affecting the work (and the universality) of our TM. On the contrary, the cell marked "Halt" *must* remain empty: the machine must halt if the string $\mathcal{S}$ starts with the marker γ (see Rem.3).

Concerning the second step of the TM's work almost any detail has been given in Rem.2; the corresponding piece of table is given by:

Table 2. The second part of the TM's work.

Direction	State	0	α	β	γ	Description
$\leftarrow$	E	$\alpha\mathcal{R}I$	$\alpha\mathcal{L}E$	$\beta\mathcal{L}F$	$\gamma\mathcal{L}E$	searches leftward
$\leftarrow$	F	$\alpha\mathcal{L}G$	$\alpha\mathcal{L}E$	$\boxed{\ }$	$\boxed{\ }$	doubt
$\leftarrow$	G	$\alpha\mathcal{R}H$	$\boxed{\ }$	$\beta\mathcal{R}J$	$\boxed{\ }$	decided
$\rightarrow$	H	$\gamma\mathcal{L}E$	$\alpha\mathcal{R}H$	$\beta\mathcal{R}H$	$\gamma\mathcal{R}H$	appends γ
$\rightarrow$	I	$\alpha\mathcal{L}E$	$\alpha\mathcal{R}I$	$\beta\mathcal{R}I$	$\gamma\mathcal{R}I$	appends α

and the third step requires just one line:

Table 3. The third part of the TM's work.

Direction	State	0	α	β	γ	Description
$\rightarrow$	J	$\boxed{\ }$	$0\mathcal{R}J$	$\beta\mathcal{R}J$	$\alpha\mathcal{R}A$	Restore

We promised a 7×4 UTM and we wrote a 10×4 table; however, according with Rem.4, nothing forbids to suppress state C, by replacing anywhere a switch to state C with a switch to state D; two other similar groupments give raise to a 7×4 UTM; see Table 4. Let us point out the major differences with respect to the 7×4 UTMs of Minsky [Min], Rogozhin [Ro2] and Robinson [Rob]. In these three TMs:

- the super–marker is given by a couple $\beta\beta$ of markers;
- almost each run of the machine swaps the markers ($\beta \longleftrightarrow \gamma$).

We choosed as the super–marker the string $\beta 0 \beta$; this choice, and the fact that we never swap the markers, allows to gain one empty cell in the table (column γ of row B; the swap of the markers would require to use such a cell for other purposes). On the other hand, our UTM halts exactly where the simulated tag–system would stop: when the machine halts, the output (to be interpreted in terms of tag–systems) starts from the γ scanned by the Head and ends on the first blank on the right.

Table 4. A 7×4 UTM with 25 commands and easy recover of the output.

Old names	New names	0	α	β	γ
A, B	T	$0\mathcal{L}T$	$0\mathcal{L}T$	$\beta\mathcal{L}U$	Halt
C, D	U	$\alpha\mathcal{R}U$	$0\mathcal{L}T$	$\beta\mathcal{R}U$	$\alpha\mathcal{L}V$
E	V	$\alpha\mathcal{R}Z$	$\alpha\mathcal{L}V$	$\beta\mathcal{L}W$	$\gamma\mathcal{L}V$
F	W	$\alpha\mathcal{L}X$	$\alpha\mathcal{L}V$	$\square$	$\square$
G, J	X	$\alpha\mathcal{R}Y$	$0\mathcal{R}X$	$\beta\mathcal{R}X$	$0\mathcal{R}T$
H	Y	$\gamma\mathcal{L}V$	$\alpha\mathcal{R}Y$	$\beta\mathcal{R}Y$	$\gamma\mathcal{R}Y$
I	Z	$\alpha\mathcal{L}V$	$\alpha\mathcal{R}Z$	$\beta\mathcal{R}Z$	$\gamma\mathcal{R}Z$

4 A 10×3 UTM

We come back to the original 10 states of Tables 1, 2 and 3 because, with respect to Table 4, some different groupment will be needed. In order to work with only 3 tape–symbols we will now replace the marker γ by the string $\beta\beta$; this choice (say "the marker in $\mathcal{S}$ is twice the marker in $\mathcal{P}$") was already used by Rogozhin in its 10×3 UTM[6]. Concerning row G in Table 2 it is sufficient to suppress the (empty) column γ; concerning row A in Table 1 we just need to (suppress the column γ and) put the Halt in column β; also for row I in Table 2 we confined ourselves to suppress the column γ (thus row I will, as needed, leave unchanged the strings $\beta\beta$); concerning rows E, F of Table 2 we just did a little change: we suppressed of course column γ in both rows, but the (empty) column β in row F needs now to be filled with $\beta\mathcal{R}E$ (thus accomplishing the work of column γ in the old row E). All remaining rows require many steps (and possibly many states; e.g. state H splits into H, H_1, H_2) for the simulation.

[6] see [Ro2], where our symbols $0, \alpha, \beta$ are denoted by $0, 1, b$ respectively

Table 5. States without horizontal separating line can be grouped together, thus getting a 10×3 UTM.

State	0	α	β
A	$\square$	$0\mathcal{L}B$	Halt
H_1	$\beta\mathcal{L}H_2$	$\square$	$\square$
B	$0\mathcal{L}B$	$0\mathcal{L}B$	$\beta\mathcal{L}C$
C	$\alpha\mathcal{R}D$	$0\mathcal{L}B$	$\square$
D	$\alpha\mathcal{R}D$	$0\mathcal{L}B$	$\beta\mathcal{R}D_1$
D_1	$\alpha\mathcal{R}D$	$\square$	$\alpha\mathcal{L}D_2$
D_2	$\square$	$\square$	$\alpha\mathcal{L}D_3$
D_3	$\square$	$\alpha\mathcal{L}E$	$\square$

State	0	α	β
E	$\alpha\mathcal{R}I$	$\alpha\mathcal{L}E$	$\beta\mathcal{L}F$
F	$\alpha\mathcal{L}G$	$\alpha\mathcal{L}E$	$\beta\mathcal{L}E$
H_2	$\square$	$\square$	$\beta\mathcal{L}E$
G	$\alpha\mathcal{R}H$	$\square$	$0\mathcal{R}J_1$
J	$\square$	$0\mathcal{R}J$	$0\mathcal{R}J_1$
J_1	$\square$	$\alpha\mathcal{L}J_2$	$0\mathcal{R}A$
J_2	$\beta\mathcal{R}J$	$\square$	$\square$
H	$\beta\mathcal{R}H_1$	$\alpha\mathcal{R}H$	$\beta\mathcal{R}H$
I	$\alpha\mathcal{L}E$	$\alpha\mathcal{R}I$	$\beta\mathcal{R}I$

Remark 5. Row H_1 could be grouped both with J or with H_2, but it is more convenient to group it with row A, and to absorbe row H_2 into (the new form of) row F. The most tricky simulations concern column β in row J, where we wrote a 0; the choice is not very natural, but allows to treat any case by just one more state (the union of J_1 and J_2). Concerning row G, as already said, it could be treated by simply suppressing the column γ; however nothing forbids of apply to it the same treatment used for row J: in column β we can write $0\mathcal{R}J_1$. The choice, already not so natural in the treatment of row J, can here seem absurd. However it works: the head, in state J_1, will meet an α that forces the TM to come back and write the needed β. In such a way row G can be joined to row J (because we did not join J and H_1); and row H_1 can be joined to row A thus giving raise to the 10×3 UTM of Table 5.

Remark 6. Concerning the structure of the whole machine, we find our TM somewhat simpler than the Rogozhin's one: we do not need to differentiate the treatment for even–length and odd–length productions; and the couple $\beta\beta$ acts simply as a marker[7].

[7] in the Rogozhin's UTM the second β of the couple acts as "first normal symbol α of the following block"; this operation being compensated by the fact that, when writing the marker $\beta\beta$ on the right of $\mathcal{S}$, the leftmost β of the couple overwrites the last α already written...

5 A 19×2 UTM

A fundamental result of Shannon [Sha] states that any Turing Machine with m states and n symbols can be simulated by a 2–symbols TM with $m \cdot \phi(n)$ states, where the function ϕ is given by:

$$\text{let } l \text{ be minimal such } n \leq 2^l; \text{ then } \phi(n) = 3 \cdot 2^l + 2l - 7.$$

Remark 7. The Shannon result is the first step for the construction of small UTMs; the second step being the fact that any 2–symbols TM can be simulated by a tag system, and the third one being that (as we shortly explained) tag systems can be simulated by a small Turing Machine. Details can e.g. found in [Rob], where is also given an easier proof (at the expenses of a worst value for ϕ) of the Shannon's result.
The first step in the Shannon's technique transforms the original tape by replacing each old symbol with its l–digits binary description[8]. If applied to 7×4 TMs the Shannon's formula gives the existence of a simulating 63–states TM; perhaps the factor $\phi(4) = 9$ in the Shannon's formula[9], one can expect that better values should be obtained by using some peculiarity of the table (empty cells, repeated commands,...); in fact, applied to the Robinson's UTM, it can be lowered to a (quite astonishing) value close to 3: we proved in [Bai] that such a UTM can be simulated by a 22×2 TM[10].

Also our 7×4 UTM can be simulated by a 22×2 TM; however, in order to prove a better result, we will follow a different strategy, using variable length codes: the marker β will be replaced by a one–digit symbol, while α, γ and 0 will be replaced by two–digits symbols.

With respect to the original 10 states (see Tables 1, 2 and 3) we will "duplicate" the old symbol α, by replacing it somewhere with an $\overrightarrow{\alpha}$ and somewhere with an $\overleftarrow{\alpha}$, thus getting a 10×5 table; the replacement being done in such a way that arrrows over the α denote the direction along which the Head moves when it arrives to scan such a symbol. It is not a difficult task: in the initial tape the symbol α appears only in $\mathcal{S}$ (thus it will be scanned "coming from the left") and we replace it by $\overrightarrow{\alpha}$; in the table commands like $\alpha\mathcal{R}K$ must be replaced by $\overleftarrow{\alpha}\mathcal{R}K$[11] while commands like $\alpha\mathcal{L}K$ must be replaced by $\overrightarrow{\alpha}\mathcal{L}K$. The result is given in Table 6.

Remark 8. In row J column γ of Table 6 we wrote "$\overleftarrow{\alpha}\mathcal{R}A$" instead of "$0\mathcal{R}A$", as in Table 3; it is quite obvious that this does not affect the behaviour of the UTM. In fact, concerning the next symbol to scan, we have two cases: it can be a γ, or a $\overrightarrow{\alpha}$. In the first case the machine will halt (thus the $\overleftarrow{\alpha}$ we wrote has no relevance); in the second case the machine will come back in state B, where symbols $\overleftarrow{\alpha}$ and 0 are treated in the same way.

[8] of course initial zeros can not be suppressed!
[9] factor which is sharp in the framework of the simulation of *any* 4–symbols TM
[10] a 22×2 UTM was also obtained, by direct construction, in Rogozhin [Ro3]
[11] when the symbol we are writing will be read again, we will arrive to it moving left!

Table 6. An intermediate 10×5 table.

State	0	$\overrightarrow{\alpha}$	$\overleftarrow{\alpha}$	β	γ
A	$\square$	$0\mathcal{L}B$	$\square$	$\square$	Halt
B	$0\mathcal{L}B$	$\square$	$0\mathcal{L}B$	$\beta\mathcal{L}C$	$\square$
C	$\overleftarrow{\alpha}\mathcal{R}D$	$\square$	$0\mathcal{L}B$	$\square$	$\square$
D	$\overleftarrow{\alpha}\mathcal{R}D$	$0\mathcal{L}B$	$\square$	$\beta\mathcal{R}D$	$\overrightarrow{\alpha}\mathcal{L}E$
E	$\overleftarrow{\alpha}\mathcal{R}I$	$\square$	$\overrightarrow{\alpha}\mathcal{L}E$	$\beta\mathcal{L}F$	$\gamma\mathcal{L}E$
F	$\overrightarrow{\alpha}\mathcal{L}G$	$\square$	$\overrightarrow{\alpha}\mathcal{L}E$	$\square$	$\square$
G	$\overleftarrow{\alpha}\mathcal{R}H$	$\square$	$\square$	$\beta\mathcal{R}J$	$\square$
H	$\gamma\mathcal{L}E$	$\overleftarrow{\alpha}\mathcal{R}H$	$\square$	$\beta\mathcal{R}H$	$\gamma\mathcal{R}H$
I	$\overrightarrow{\alpha}\mathcal{L}E$	$\overleftarrow{\alpha}\mathcal{R}I$	$\square$	$\beta\mathcal{R}I$	$\gamma\mathcal{R}I$
J	$\square$	$0\mathcal{R}J$	$\square$	$\beta\mathcal{R}J$	$\overleftarrow{\alpha}\mathcal{R}A$

With respect to Table 6, let us now rename the five symbols 0, $\overrightarrow{\alpha}$, $\overleftarrow{\alpha}$, β and γ by $''00''$, $''01''$, $''10''$, $''1''$ and $''11''$ respectively; the next step consists in interpreting such new symbols as a single–digit symbol (the $''1''$) or as 2–digit symbols (the remaining ones[12]). Let us point out the main reason for our choice of variable–length codes: independently from the direction we are walking in, when we scan a 0 we are sure it is the starting of a normal symbol; conversely, when scanning a 1, it must be a marker (or possibly the beginning of a marker). With such a remark in mind, it is nothing but a tedious work to check that Table 7 simulates Table 6.

Let us just add a few words about the treatment of state G. In such a state we can meet only a 1 or a 00; in the first case we must (and we do, with the command $1\mathcal{R}J$) leave the 1 unchanged, going right in state J; in the second case we should replace 00 by 10, going right in state H; the command $1\mathcal{L}H$ we wrote, after two more TM's steps, has exactly this effect.

References

Bai. Baiocchi, C.: $3N + 1$, UTM e Tag–Systems. Dipartimento di Matematica dell'Università "La Sapienza" di Roma **98/38** (1998).

Min. Minsky, M.: Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ, 1967.

[12] in particular: γ is again replaced by $\beta\beta\ldots$

Table 7. States in the rightmost part can be replaced as indicated; joining states C and J_1 we get a 19×2 UTM.

State	0	1	State	0	1	State	0	1
A	$1\mathcal{R}A_1$	Halt				$A_1 \to D_1$	$\square$	$0\mathcal{L}B_1$
B	$0\mathcal{L}B_1$	$1\mathcal{L}C$	B_1	$0\mathcal{L}B$	$0\mathcal{L}B$			
C	$0\mathcal{L}C_1$	$\square$	C_1	$1\mathcal{R}C_2$	$0\mathcal{L}B$	$C_2 \to D_1$	$0\mathcal{R}D$	$\square$
D	$1\mathcal{R}D_1$	$1\mathcal{R}D_2$	D_1	$0\mathcal{R}D$	$0\mathcal{L}B_1$			
			D_2	$1\mathcal{R}D_1$	$1\mathcal{L}D_3$	$D_3 \to E_1$	$\square$	$0\mathcal{L}E$
E	$1\mathcal{L}E_1$	$1\mathcal{L}E_3$	E_1	$1\mathcal{R}E_2$	$0\mathcal{L}E$	$E_2 \to I_1$	$\square$	$0\mathcal{R}I$
			E_3	$1\mathcal{L}F_1$	$1\mathcal{L}E$			
			F_1	$0\mathcal{L}G$	$0\mathcal{L}E$	$F \to E_3$	$1\mathcal{L}F_1$	$\square$
G	$1\mathcal{L}H$	$1\mathcal{R}J$						
H	$1\mathcal{R}H_1$	$1\mathcal{R}H$	H_1	$1\mathcal{L}H_2$	$0\mathcal{R}H$	$H_2 \to E_3$	$\square$	$1\mathcal{L}E$
I	$1\mathcal{R}I_1$	$1\mathcal{R}I$	I_1	$1\mathcal{L}I_2$	$0\mathcal{R}I$	$I_2 \to E_1$	$\square$	$0\mathcal{L}E$
J	$0\mathcal{R}J_1$	$1\mathcal{R}J_2$	J_1	$\square$	$0\mathcal{R}J$			
			J_2	$0\mathcal{R}J_1$	$0\mathcal{R}A$			

Rob. Robinson, R. M.: Minsky's small Turing machine. International Journal of Mathematics **2 (5)** (1991) 551–562.

Ro1. Rogozhin, Y.: Seven universal Turing machine. Systems and Theoretical Programming, Mat.Issled **69** Academiya Nauk Moldavskoi SSR, Kishinev (1982) 76–90 (Russian).

Ro2. Rogozhin, Y.: Small universal Turing machines. Theoretical Computer Science **168-2** (1996) 215–240.

Ro3. Rogozhin, Y.: A universal Turing machine with 22 states and 2 symbols. Romanian Journal of Information Science and Technology **1** (3) (1998) 259–265.

Sha. Shannon, C. E.: A Universal Turing Machine With Two Internal States. Automata Studies (*Shannon & McCarthy Editors*) Princeton University Press (1956) 157–165.

Computation in Gene Networks

Asa Ben-Hur[1] and Hava T. Siegelmann[2]

[1] BioWulf Technologies
2030 Addison st. suite 102, Berkeley, CA 94704
[2] Lab for Inf. & Decision Systems
MIT Cambridge, MA 02139, USA
iehava@ie.technion.ac.il

Abstract. We present a biological computing paradigm that is based on genetic regulatory networks. Using a model of gene expression by piecewise linear differential equations we show that the evolution of protein concentrations in a cell can be considered as a process of computation. This is demonstrated by showing that this model can simulate memory bounded Turing machines. The simulation is robust with respect to perturbations of the system, an important property for both analog computers and biological systems.

1 Introduction

In recent years scientists have been looking for new paradigms for constructing computational devices. These include quantum computation [1], DNA computation [2], neural networks [3], neuromorphic engineering [4] and other analog VLSI devices. This paper describes a new paradigm based on genetic regulatory networks. The concept of a "genetic network" refers to the complex network of interactions between genes and gene products in a cell [5]. Since the 60's genetic regulatory systems are thought of as "circuits" or "networks" of interacting components [6], and were described in computer science terms: The genetic material is the "program" that guides protein production in a cell; protein levels determine the evolution of the network at subsequent times, and thus serve as its "memory". This analogy between computing and the process of gene expression was pointed out in various papers [7,8]. Bray suggests that protein based circuits are the device by which unicellular organisms react to their environment, instead of a nervous system [7]. However, until recently this was only a useful metaphor for describing gene networks. The papers [9,10] describe the successful fabrication of synthetic networks, i.e. *programming* of a gene network. In this paper we compare the power of this computational paradigm with the standard digital model of computation. In a related series of papers it is shown both theoretically and experimentally that chemical reactions can be used to implement Boolean logic and neural networks (see [11] and references therein).

Protein concentrations are continuous variables that evolve continuously in time. Moreover, biological systems do not have timing devices, so a description in terms of a *map* that simultaneously updates the system variables is inadequate. It

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 11–24, 2001.

is thus necessary to model gene networks by differential equations, and consider them as analog computers. The particular model of genetic networks we analyze here assumes switch-like behavior, so that protein concentrations are described by piecewise linear equations (see equation (1) below) [12,8], but we believe our results to hold for models that assume sigmoid response (see discussion). These equations were originally proposed as models of chemical oscillations in biological systems [13]. In this paper we make the analogy between gene networks and computational models complete by formulating an abstract computational device on the basis of these equations, and showing that this analog model can simulate a computation of a Turing machine [14]. The relation between digital models of computation and analog models is explored in a recent book [15], mainly from the perspective of neural networks. It is shown there that analog models are potentially stronger than digital ones, assuming an ideal noiseless environment. In this paper on the other hand we consider the possibility of noise and propose a design principle which makes the model equations robust. This comes at a price: we can only simulate memory bounded Turing machines. However, we argue that any physically realizable robust simulation has this drawback. On the subject of computation in a noisy environment see also [16,17]. We found that the gene network proposed in [9] follows this principle, and we quote from that paper their statement that "theoretical design of complex and practical gene networks is a realistic and achievable goal".

Computation with biological hardware is also the issue in the field of DNA computation [2]. As a particular example, the guided homologous recombination that takes place during gene rearrangement in ciliates was interpreted as a process of computation [18]. This process, and DNA computation in general, are symbolic, and describe computation at the molecular level, whereas gene networks are analog representations of the macroscopic evolution of protein levels in a cell.

2 Piecewise Linear ODEs for Gene Networks

In this section we present model equations for gene networks, and note a few of their dynamical properties [19]. The concentration of N proteins (or other biochemicals in a more general context) is given by N non-negative real variables $y_1, \ldots, y_N$. Let $\theta_1, \ldots, \theta_N$ be N threshold concentrations. The production rate of each protein is assumed to be constant until a threshold is crossed, when the production rate assumes a new value. This is expressed by the equations:

$$\frac{dy_i}{dt} = -k_i y_i + \tilde{\Lambda}_i(Y_1, \ldots, Y_N) \ , \tag{1}$$

where Y_i is a Boolean variable associated with y_i, equal to 1 if $y_i \geq \theta_i$ and 0 otherwise; k_i is the degradation rate of protein i and $\tilde{\Lambda}_i$ is its production rate when gene i is "on". These equations are a special case of the model of Mestl et. al. [12] where each protein can have a number of thresholds, compared with just one threshold here (see also [20]). When there is just one threshold it is easy to

associate Boolean values with the continuous variables. For simplicity we take $k_i = 1$, and define

$$x_i = y_i - \theta_i,$$

with the associated Boolean variables

$$X_i = \mathrm{sgn}(x_i),$$

where $\mathrm{sgn}(x) = 1$ for $x \geq 0$ and zero otherwise. Also denote $\Lambda_i(X_1, \ldots, X_N) = \tilde{\Lambda}_i(Y_1, \ldots, Y_N) - \theta_i$. Equation (1) now becomes:

$$\frac{dx_i}{dt} = -x_i + \Lambda_i(X_1, \ldots, X_N)\,, \tag{2}$$

and Λ_i is called the *truth table*. The set in $\mathbb{R}^N$ which corresponds to a particular value of a Boolean vector $X = (X_1, \ldots, X_N)$ is an *orthant* of $\mathbb{R}^N$. By abuse of notation, an orthant of $\mathbb{R}^N$ will be denoted by a Boolean vector X. The trajectories in an orthant are straight lines directed to a focal point $\Lambda(X) = (\Lambda_1(X), \ldots, \Lambda_N(X))$, as seen from equation (3) below. If the focal point $\Lambda(X)$ at a point $x = (x_1, \ldots, x_N)$ is in the same orthant as x, then the dynamics converges to the focal point. Otherwise it crosses the boundary to another orthant, where it is redirected to a different focal point. The sequence of orthants $X(1), X(2), \ldots$ that correspond to a trajectory $x(t)$ is called the *symbolic dynamics* of the vector field. In the next sections we will associate the symbolic dynamics of a a model Gene Network (GN) with a process of computation.

2.1 Dynamics

In an orthant all the Λ_i are constant, and the equations (2) are easily integrated. Starting from a point $x(0)$,

$$x_i(t) = \lambda_i + (x_i(0) - \lambda_i)e^{-t}\,, \tag{3}$$

where $\lambda_i = \Lambda_i(X_1(0), \ldots, X_N(0))$. The time t_i at which the hyper-plane $x_i = 0$ is crossed is given by

$$t_i = \ln \frac{\lambda_i - x_i(0)}{\lambda_i}$$

The switching time t_s is the time it takes to reach a new orthant:

$$t_s = \min_i t_i$$

We will consider networks with $\Lambda_i = \pm 1$ in all orthants. Thus when the focal point is in a different orthant than $x(0)$ we have that

$$t_s \leq \ln 2\,. \tag{4}$$

This gives a criterion for determining whether a GN is converging to a fixed point: if the time from the last switching time is larger than $\ln 2$, then the system is converging to the current focal point.

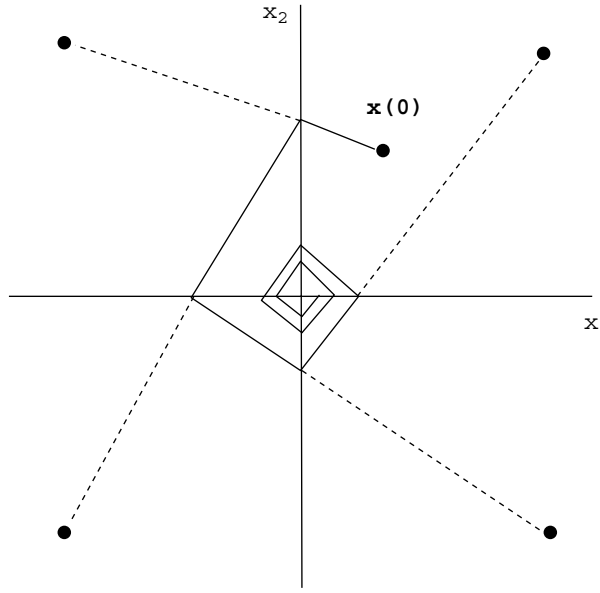


Fig. 1. A piecewise linear flow in two dimensions.

A network as in (2) is a continuous time version of a discrete network:

$$Z_i \;\mapsto\; \mathrm{sgn}(\Lambda_i(Z)) \;, \tag{5}$$

where $Z \in \{0,1\}^N$ is a vector of Boolean variables. This dynamics does not necessarily visit the same sequence of orthants as the corresponding continuous network.

We want to simulate a discrete dynamical system (Turing machine) with a continuous time GN. To bridge the gap, we first construct a discrete gene network of the form (5), whose corresponding continuous GN has the same symbolic dynamics. We will use truth tables with the following property:

Definition 1. *Two orthants are said to be* adjacent *if they differ in exactly one coordinate. A truth table Λ will be called* adjacent *if $\Lambda(X)$ is in the same orthant as X or in an adjacent orthant to X for all $x \in \mathbb{R}^N$. A network with an adjacent truth table will also be called adjacent.*

We note that in an adjacent network, all initial conditions in an orthant lead to the same adjacent orthant. Therefore all initial conditions of (2) in the same orthant have the same symbolic dynamics. An immediate result is the following lemma:

Lemma 1. *Let Λ be an adjacent truth table, then the symbolic dynamics of a continuous GN is the same as the dynamics of its discrete counterpart (5): $X(k) = Z(k),\ k = 0, 1, \ldots$, where $Z(k)$ is the kth iterate of (5) and $X(k)$ is the kth orthant visited by (2), for every initial condition of (2) corresponding to Z_0.*

In view of the above discussion, the focal points and the initial condition of an adjacent network can be taken as points in $\{-1,1\}^N$, and the problem of constructing a continuous network that simulates a discrete dynamics is reduced to the problem of constructing a discrete network that changes only one variable at each iteration.

When the truth table is not adjacent, a discrete network may show qualitatively different dynamics than its continuous counterpart: continuous high-dimensional GN's are "typically" chaotic [21,22]. However chaos is a dynamical behavior that is impossible in dynamical systems with a finite state space. The lack of sensitivity of the dynamics of an adjacent GN to the placement of the initial condition, and its equivalence to a discrete network leads to the following statement:

Corollary 1. *Adjacent networks are not chaotic.*

This implies a measure of stability to the dynamics of adjacent networks. Lack of sensitivity to perturbations of the system is captured by the following property of adjacent networks:

Claim. Let Λ be an adjacent truth table, and let $\tilde{\Lambda}$ be a truth table whose entries are $\Lambda_i(X) + \delta_i(X)$, where $\delta_i(X) \in [-c, c]$ for some $1 > c > 0$. Then $\tilde{\Lambda}$ is also adjacent, and the two networks have the same symbolic dynamics.

The robustness of adjacent networks is an important property for both analog computers and biological systems. The robustness of biological systems leads us to speculate that adjacency might be a principle underlying the robust behavior in the modeled systems. Adjacency can also be taken as a principle which can be used in the design of robust networks.

3 Preliminaries

In this section we give the definition of the Turing machine that will be simulated and provide the relevant concepts from complexity theory [14].

Definition 2. *A Turing machine is a tuple* $M = (K, \Sigma, \Gamma, \delta, Q_1, Q_q)$. $K = \{Q_1, \ldots, Q_q\}$ *is a finite set of* states*;* $Q_1, Q_q \in K$ *are the* initial/halting *states respectively;* Σ *is the* input alphabet*;* $\Gamma = \Sigma \cup \{blank, \#\}$ *is the* tape alphabet *which includes the* blank symbol *and the left end symbol,* #*, which marks the end of the tape;* $\delta : K \times \Gamma \to K \times \Sigma \times \{L, R\}$ *is the* transition function*, and* L/R *signify left/right movement of the read-write head. The transition function is such that it cannot proceed left of the left end marker, and does not erase it, and every tape square is blank until visited by the read-write head.*

At the beginning of a computation an input sequence is written on the tape starting from the square to the right of the left-end marker. The head is located at the leftmost symbol of the input string, and the finite-control is initialized at its start state Q_1. The computation then proceeds according to the transition function, until reaching the halting state. Without loss of generality we suppose

that the input alphabet, Σ, is $\{0, 1\}$. We say that a Turing machine *accepts* an input word w if the computation on input w reaches the halting state with "1" written on the tape square immediately to the right of the left-end marker. If the machine halts with "0" at the first tape square, then the input is rejected by the machine. Other conventions of acceptance are also common. The language of a Turing machine M is the set $L(M)$ of strings accepted by M.

Languages can be classified by the computational resources required by a Turing machine which accepts them. The classification can be according to *time* or *space* (memory). The time complexity of a computation is the number of steps until halting and its space complexity is the number of tape squares used in the computation. The complexity classes P and PSPACE are defined to be the classes of languages that are accepted in polynomial time and polynomial space, respectively. More formally, a language L is in PSPACE if there exists a Turing machine M which accepts L, and there exists $c > 0$ such that on all inputs of length n M accesses $O(n^c)$ tape squares. To make a finer division one defines the class SPACE($s(n)$) of languages that are accepted in space $s(n)$.

3.1 On the Feasibility of Turing Machine Simulation

Turing machine simulations by differential equations appear in a number of papers: in [23] it was shown that an ODE in four dimensions can simulate a Turing machine, but the explicit form of the ODE is not provided. Finite automata and Turing machines are simulated in [24] by piecewise constant ODEs. Their method is related to the one used here.

A Turing machine has a countably infinite number of configurations. In order to simulate it by an ODE, an encoding of these configurations is required. These can essentially be encoded into a continuum in two ways:

- in a bounded set, and two configurations can have encodings that are arbitrarily close;
- in an unbounded set, keeping a minimal distance between encodings.

In the first possibility, arbitrarily small noise in the initial condition of the simulating system may lead to erroneous results, and is therefore unfeasible (this point is discussed in [25] in the context of simulating a Turing machine by a map in $\mathbb{R}^2$). The second option is also unfeasible since a physical realization must be finite in its extent. Thus simulation of an arbitrary Turing by a realizable analog machine is not possible, and simulation of resource bounded Turing machines is required. In this chapter we will show a robust simulation of memory bounded Turing machines by adjacent GN's.

4 Computing with GN's

In this section we formulate GN's as computational machines. The properties of adjacent networks suggest a natural interpretation of an orthant X as a robust representation of a discrete configuration. The symbolic dynamics of a network,

i.e. the values $X(t)$ will be interpreted as a series of configurations of a discrete computational device.

Next we specify how it receives input and produces an output. A subset of the variables will contain the input as part of the initial condition of the network, and the rest of the variables will be initialized in some uniform way. Since we are considering $\Sigma = \{0, 1\}$, the input is encoded into the binary values $X(0)$. To specify a specific point in the orthant X we choose -1 to correspond to 0 and 1 to correspond to 1. Note that for adjacent networks *any* value of $x(0)$ corresponding to $X(0)$ leads to the same computation, so this choice is arbitrary.

In addition to input variables, another subset of the variables is used as output variables. For the purpose of language accepting a single output variable is sufficient. There are various ways of specifying halting. One may use a dynamical property, namely convergence to a fixed point, as a halting criterion. We noted that convergence to a fixed point is identified when after a time $\ln 2$ no switching has occurred (see equation (4)). Non-converging dynamics correspond to non-halting computations. While such a definition is natural from a dynamics point of view, it is not biologically plausible, since a cell evolves continuously, and convergence to a fixed point has the meaning of death. Another approach is to set aside a variable that will signify halting. Let this variable be X_N. It is initially set to 0, and when it changes to 1, the computation is complete, and the output may be read from the output variables. In this case non-halting computations are trajectories in which X_N never assumes the value 1. After X_N has reached the value 1, it may be set to 0, and a new computational cycle may begin. A formal definition of a GN as a computational machine is as follows.

Definition 3. *A genetic network is a tuple $G = (V, I, O, \Lambda, x_0, X_N)$, where V is the set of variables indexed by $\{1, \ldots, N\}$; $I \subseteq V$, $|I| = n$ and $O \subseteq V$, $|O| = m$ are the set of* input *and* output *variables respectively; $\Lambda : \{0, 1\}^N \to \{-1, 1\}^N$ is a truth table for a flow (2); $x_0 \in \{-1, 1\}^{N-n}$ is the initialization of variables in $V \setminus I$. X_N is the halting variable that is initialized to 0. A computation is halted the first time that $X_N = 1$.*

A GN G with n input variables and m output variables computes a partial mapping

$$f_G : \{0, 1\}^n \to \{0, 1\}^m,$$

which is the value of X_i for $i \in O$ when the halting state $X_N = 1$ is reached. If on input w the net does not reach a halting state then $f_G(w)$ is undefined.

We wish to characterize the languages accepted by GN's. For this purpose it is enough to consider networks with a single output variable, and say that a GN G *accepts* input $w \in \{0, 1\}^*$ if $f_G(w) = 1$. A fixed network has a constant number of input variables. Therefore it can only accept languages of the form

$$L_n = L \cap \{0, 1\}^n \ , \tag{6}$$

where $L \subseteq \{0, 1\}^*$. To accept a language which contains strings of various lengths we consider a family $\{G_n\}_{n=1}^{\infty}$ of networks, and say that such a family accepts a language L if for all n, $L(G_n) = L_n$.

Before we define the complexity classes of GN's we need to introduce the additional concept of *uniformity*. A GN with N variables is specified by the entries of its truth table Λ. This table contains the 2^N focal points of the system. Thus the encoding of a general GN requires an exponential amount of information. In principle one can use this exponential amount of information to encode *every* language of the form $L \cap \{0,1\}^n$, and thus a series of networks exists for every language $L \subseteq \{0,1\}^*$. Turing machines on the other hand, accept only the subset of *recursive* languages [14]. A Turing machine is finitely specified, essentially by its transition function, whereas the encoding of an infinite series of networks is not necessarily finite. To obtain a series of networks that is equivalent to a Turing machine it is necessary to impose the constraint that truth tables of the series of networks should all be created by one finitely encoded machine. The device which computes the truth table must be simple in order to demonstrate that the computational power of the network is not a by-product of the computing machine that generates its transition function, but of the complexity of its time evolution. We will use a finite automaton with output [26], which is equivalent to a Turing machine which uses constant space. The initial condition of the series of networks needs also to be computable in a uniform way, in the same way as the truth table, since it can be considered as "advice", as in the model of advice Turing machines [14]. We now define:

Definition 4. *A family of networks* $\{G_n\}_{n=1}^{\infty}$ *is called* uniform *if there exist constant memory Turing machines* M_1, M_2 *that compute the truth table and initial condition as follows: on input* $X \in \{0,1\}^n$ M_1 *outputs* $\Lambda(X)$*;* M_2 *outputs* x_0 *for* G_n *on input* 1^n *..*

With the definition of uniformity we can define complexity classes. Given a function $s(n)$, we define the class of languages which are accepted by networks with less than $s(n)$ variables (not including the input variables):

$$GN(s(n)) = \{L \subseteq \{0,1\}^* \mid \text{there exists a uniform family of networks } \{G_n\}_{n=1}^{\infty}, \text{ s.t. } L(G_n) = L_n \text{ and } N \leq s(n) + n\}$$

The class with polynomial $s(n)$ will be denoted by PGN. The classes Adjacent-GN($s(n)$) and Adjacent-PGN of adjacent networks are similarly defined. Time constrained classes can also be defined.

5 The Computational Power of GN's

In this section we outline the equivalence between memory-bounded Turing machines and networks with adjacent truth tables. We begin with the following lemma:

Lemma 2. *Adjacent-GN*$(s(n)) \subseteq$ *SPACE*$(s(n))$.

Proof. Let $L \in$ Adjacent-GN($s(n)$). There exists a family of adjacent GN's, $\{G_n\}_{n=1}^{\infty}$ with truth tables $\Lambda^{(n)}$, such that $L(G_n) = L_n$, and constant memory

Turing machines M_1, M_2 that compute $\Lambda^{(n)}$ and x_0. Since $\Lambda^{(n)}$ is an adjacent truth table, and we are only interested in its symbolic dynamics, we can use the corresponding discrete network. The Turing machine M' for L will run M_1 to obtain the initial condition, and then run M_2 to generate the iterates of the discrete network. M' will halt when the network has reached a fixed point. The network G_n has $s(n)$ variables on inputs of length n, therefore the simulation requires at least that much memory. It is straightforward to verify that $O(s(n))$ space is also sufficient.

The Turing machine simulation in the next section shows:

Lemma 3. *Let M be a Turing machine working in space $s(n)$, then there exists a sequence of uniform networks $\{G_n\}_{n=1}^{\infty}$ with $s(n)$ variables such that $L(G_n) = L(M) \cap \{0,1\}^n$.*

We conclude:

Theorem 1. *Adjacent-GN($s(n)$)=SPACE($s(n)$).*

As a result of claim 2.1 we can state that adjacent networks compute robustly. This is unlike the case of dynamical systems which simulate arbitrary Turing machines, where arbitrarily small perturbations of a computation can corrupt the result of a computation (see e.g. [24,27,15]). The simulation of memory bounded machines is what makes the system robust. The above theorem gives only a lower bound on the computational power of the general class of GN's, i.e.:

Corollary 2. *SPACE($s(n)$) $\subseteq$ GN($s(n)$).*

One can obtain non-uniform computational classes in two ways, either by allowing advice to appear as part of the initial condition, or by using a weaker type of uniformity for the truth table. This way one can obtain an equivalence with a class of the type PSPACE/poly [14].

6 Turing Simulation by GN's

Since the discrete and continuous networks associated with an adjacent truth table have the same symbolic dynamics, it is enough to describe the dynamics of an adjacent discrete network. We show how to simulate a space bounded Turing machine by a discrete network whose variables represent the tape contents, state and head position. An adjacent map updates a single variable at a time. To simulate a general Turing machine by such a map each computational step is broken into a number of operations: updating the tape contents, moving the head, and updating the state; these steps are in turn broken into steps that can be performed by single bit updates.

Let M be a Turing machine that on inputs of length n uses space s. Without loss of generality we suppose that the alphabet of the Turing machine is $\Sigma = \{0,1\}$. To encode the three symbols $\{0,1,blank\}$ by binary variables we use a pair of variables for each symbol. The first variable of the pair is zero

iff the corresponding tape position is blank; "0" is encoded as "10" and "1" is encoded as "11". Note that the left end marker symbol need not be encoded since the variables of the network have numbers. We construct a network G with variables $Y_1, \ldots, Y_s; B_1, \ldots, B_s; P_1, \ldots, P_s; Q_1, \ldots, Q_q$ and auxiliary variables $B, Y, Q'_1, \ldots, Q'_q, C_1, \ldots, C_4$. The variables

$$Y_1, \ldots, Y_s; B_1, \ldots, B_s$$

store the contents of the tape: B_i indicates whether the square i of the tape is blank or not and Y_i is the binary value of a non-blank square. The input is encoded into the variables $Y_1, \ldots, Y_n, B_1, \ldots, B_n$. Since the Turing machine signifies acceptance of an input by the value of its left-most tape square, we take the output to be the value of Y_1. The position of the read-write head is indicated by the variables

$$P_1, \ldots, P_s$$

If the head is at position i then $P_i = 1$ and the rest are zero. The state of the machine will be encoded in the variables

$$Q_1, \ldots, Q_q$$

where Q_1 is the initial state and Q_q is the halting state of the machine. State i will be encoded by $Q_i = 1$ and the rest zero.

As mentioned above, a computation of the Turing machine is broken into a number of single bit updates. After updating variables related to the state or the symbol at the head position, information required to complete the computation step is altered. Therefore we need the following temporary variables

- Y,B - the current symbol
- $Q'_1, \ldots, Q'_q$ - a copy of the current state variables.

A computation step of the Turing machine is simulated in four stages:

1. Update the auxiliary variables $Y, B, Q'_1, \ldots, Q'_q$ with the information required for the current computation step;
2. Update the tape contents;
3. Move the head;
4. Update the state.

We keep track of the stage at which the simulation is at with a set of variables $C_1, \ldots, C_4$ which evolve on a cycle which corresponds to the cycle of operations (1)-(4) above. Each state of the cycle is associated with an update of a single variable. After a variable is updated the cycle advances to its next state. However, since a variable update does not always change the value of a variable, e.g. the machine does not have to change the symbol at the head position, and since the GN needs to advance to a nearby orthant or else it enters into a fixed point each update is of the form:

```
If the variable pointed to by the cycle
   subnetwork needs updating - update it
else
   advance to the next state of the cycle
```

Suppose that the head of M is at position i and state j, and that in the current step Y_i is changed to value Y_i^+ which is a non-blank, the state is changed to state k, and the head is moved to position $i+1$. The sequence of variable updates with the corresponding cycle states is as follows:

cycle state	variable update		
0000	Y	$\leftarrow Y_i$	
0001	B	$\leftarrow B_i$	
0011	Q'_j	$\leftarrow 1$	
0111	Y_i	$\leftarrow Y_i^+$	update variable at head
0110	B_i	$\leftarrow 1$	Y_i^+ non-blank
1110	P_{i+1}	$\leftarrow 1$	new position of head
1111	P_i	$\leftarrow 0$	erase old position of head
1011	Q_k	$\leftarrow 1$	new state
1001	Q_j	$\leftarrow 0$	erase old state
1000	Q'_j	$\leftarrow 0$	prepare for next cycle

At the end of a cycle a new cycle is initiated.

On input $w = w_1 w_2 \dots w_n$, $w_i \in \{0,1\}$ the system is initialized as follows:

Y_i	$=$	$w_i,\ i = 1, \dots, s$
B_i	$=$	$1,\ i = 1, \dots, s$
P_1	$=$	1
Q_1	$=$	1
all other variables:		0

This completes the definition of the network. The computation of the initial condition is trivial, and computing the truth table of this network is essentially a bit by bit computation of the next configuration of the simulated Turing machine, which can be carried out in constant space. The network we have defined has $O(s)$ variables, and each computation step is simulated by no more than 20 steps of the discrete network. □

Remark 1. It was pointed out that the Hopfield neural network is related to GN's [28]. Theorem 1 can be proved using complexity results about asymmetric and Hopfield networks found in [29,30]. However, in this case it is harder to define uniformity, and the direct approach taken here is simpler.

7 Language Recognition vs. Language Generation

The subclass of GN's with adjacent truth-tables has relatively simple dynamics whose attractors are fixed points or limit cycles. It is still unknown whether non-adjacent GN's are computationally stronger than adjacent GN's. General GN's can have chaotic dynamics, which are harder to simulate with Turing machines. We have considered GN's as *language recognizers*: where the input arrives at the beginning of a computation, and the decision to accept or reject is based on its state when a halting state is reached. In the context of language recognition,

chaotic dynamics does not add computational power: given a chaotic system that accepts a language, there is a corresponding system that does not have a chaotic attractor for inputs on which the machine halts; such a system is obtained e.g., by defining the halting states as fixed points. However, one can also consider GN's as *language generators* by viewing the symbolic dynamics of these systems as generating strings of some language. In this case a single GN can generate strings of arbitrary length. But even in this case chaos is probably not very "useful", since the generative power of structurally stable chaotic systems is restricted to the simple class of regular languages [31]. More complex behavior can be found in dynamical systems at the onset of chaos (see [32,33]). Continuously changing the truth table can lead to a transition to chaotic behavior [34]. At the transition point complex symbolic dynamics can be expected, behavior which is not found in discrete Boolean networks.

8 Discussion

In this paper we formulated a computational interpretation of the dynamics of a switch-like ODE model of gene networks. We have shown that such an ODE can simulate memory bounded Turing machines. While in many cases such a model provides an adequate description, more realistic models assume sigmoidal response. In the neural network literature it is proven that sigmoidal networks with a sufficiently steep sigmoid can simulate the dynamics of switch-like networks [15,35]. This suggests that the results presented here carry over to sigmoidal networks as well.

The property of adjacency was introduced in order to reduce a continuous gene network to a discrete one. We consider it as more than a trick, but rather as a strategy for fault tolerant programming of gene networks. In fact, the genetic toggle switch constructed in [9] has this property. However, when it comes to non-synthetic networks, this may not be the case - nature might have other ways of programming them, which are not so transparent.

References

1. M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.
2. L. Kari. DNA computing: the arrival of biological mathematics. *The mathematical intelligencer*, 19(2):24–40, 1997.
3. J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City, 1991.
4. C. Mead. *Analog VLSI and Neural Systems.* Addison-Wesley, 1989.
5. H. Lodish, A. Berk, S.L. Zipursky, P. Matsudaira, D. Baltimore, and J. Darnell. *Molecular cell biology.* W.H. Freemand and Company, 4th edition, 2000.
6. S.A. Kauffman. Metabolic stability and epigenesis in randomly connected nets. *Journal of Theoretical Biology*, 22:437, 1969.
7. D. Bray. Protein molecules as computational elements in living cells. *Nature*, 376:307–312, July 1995.

8. H.H. McAdams and A. Arkin. Simulation of prokaryotic genetic circuits. *Annual Review of Biophysics and Biomolecular Structure*, 27:199–224, 1998.
9. T.S. Gardner, C.R. Cantor, and J.J. Collins. Construction of a genetic toggle switch in E. coli. *Nature*, 403:339–342, January 2000.
10. M.B. Elowitz and S. Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403:335–338, January 2000.
11. A. Arkin and J. Ross. Computational functions in biochemical reaction networks. *Biophysical Journal*, 67:560–578, 1994.
12. T. Mestl, E. Plahte, and S.W. Omholt. A mathematical framework for describing and analyzing gene regulatory networks. *Journal of Theoretical Biology*, 176:291–300, 1995.
13. L. Glass. Combinatorial and topological method in chemical kinetics. *Journal of Chemical Physics*, 63:1325–1335, 1975.
14. C. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass., 1995.
15. H.T. Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing Limit*. Birkhauser, Boston, 1999.
16. H.T. Siegelmann, A. Roitershtein, and A. Ben-Hur. Noisy neural networks and generalizations. In *Proceedings of the Annual Conference on Neural Information Processing Systems 1999 (NIPS*99)*. MIT Press, 2000.
17. W. Maass and P. Orponen. On the effect of analog noise in discrete time computation. *Neural Computation*, 10(5):1071–1095, 1998.
18. L.F. Landweber and L. Kari. The evolution of cellular computing: nature's solution to a computational problem. In *Proceedings of the 4th DIMACS meeting on DNA based computers*, pages 3–15, 1998.
19. L. Glass and J.S. Pasternack. Stable oscillations in mathematical models of biological control systems. *Journal of Mathematical Biology*, 6:207–223, 1978.
20. R.N. Tchuraev. A new method fo rthe analysis of the dynamics of the molecular genetic control systems. I. description of the method of generalized threshold models. *Journal of Theoretical Biology*, 151:71–87, 1991.
21. T. Mestl, R.J. Bagley, and L. Glass. Common chaos in arbitrarily complex feedback networks. *Physical Review Letters*, 79(4):653–656, 1997.
22. L. Glass and C. Hill. Ordered and disordered dynamics in random networks. *Europhysics Letters*, 41(6):599–604, 1998.
23. M.S. Branicky. Analog computation with continuous ODEs. In *Proceedings of the IEEE Workshop on Physics and Computation*, pages 265–274, Dallas, TX, 1994.
24. E. Asarin, O. Maler, and A. Pnueli. Reachability analysis of dynamical systems with piecewise-constant derivatives. *Theoretical Computer Science*, 138:35–66, 1995.
25. A. Saito and K. Kaneko. Geometry of undecidable systems. *Prog. Theor. Phys.*, 99:885–890, 1998.
26. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
27. P. Koiran and C. Moore. Closed-form analytic maps in one and two dimensions can simulate universal Turing machines. *Theoretical Computer Science*, 210:217–223, 1999.
28. J.E. Lewis and L. Glass. Nonlinear dynamics and symbolic dynamics of neural networks. *Neural Computation*, 4:621–642, 1992.
29. P. Orponen. The computational power of discrete hopfield nets with hidden units. *Neural Computation*, 8:403–415, 1996.

30. P. Orponen. Computing with truly asynchronous threshold logic networks. *Theoretical Computer Science*, 174:97–121, 1997.
31. C. Moore. Generalized one-sided shifts and maps of the interval. *Nonlinearity*, 4:727–745, 1991.
32. J.P. Crutchfield and K. Young. Computation at the onset of chaos. In W.H. Zurek, editor, *Complexity, Entropy and the Physics of Information*, pages 223–269, Redwood City, CA, 1990. Addison-Wesley.
33. C. Moore. Queues, stacks, and transcendentality at the transition to chaos. *Physica D*, 135:24–40, 2000.
34. R. Edwards, H.T. Siegelmann, K. Aziza, and L. Glass. Symbolic dynamics and computation in model gene networks. in preparation.
35. J. Sima and P. Orponen. A continuous-time hopfield net simulation of discrete neural networks. Technical Report 773, Academy of Sciences of the Czech Republic, 1999.

Power, Puzzles and Properties of Entanglement

Jozef Gruska[1,2,⋆] and Hiroshi Imai[1]

[1] ERATO project, University of Tokyo,
Hongon 5-28-3, Bunkyo-ku, Tokyo 113-0033, Japan,
[2] Faculty of Informatics, Masaryk University,
Botanická 68a, 602 00 Brno, Czech Republic
gruska@informatics.muni.cz

Abstract. Quantum entanglement is behind various puzzling/mysterious or paradoxical phenomena of quantum mechanics in general and of quantum information processing and communication (QIPC) in particular. Or, using less strong terms, quantum entanglement leads to highly nonintuitive or even counterintuitive effects. Some of them have been known for long time. New are being discovered almost daily. At the same time quantum entanglement is considered to be a very powerful resource for QIPC and, together with quantum superposition, quantum parallelism and quantum measurement, to be the main resource of the (provable) exponentially larger power of QIPC with respect to the classical information processing and communication.

It is increasingly realized that quantum entanglement is at the heart of quantum physics and as such it may be of very broad importance for modern science and future technologies. It has also been to a large extent quantum entanglement that has led in quantum mechanics to asking fundamentally new questions and has brought a chance to see in a more profound way various features of the quantum world (and this way it can contribute to a new understanding of the foundations of quantum mechanics). There is therefore a large need to understand, analyze and utilize this resource.

The aim of the paper is, on one side, to illustrate and analyze various puzzling phenomena that are due to the quantum entanglement and related nonlocality of quantum mechanics, as well as computational and, especially, communicational power of quantum entanglement. On the other side, the aim of the paper is to discuss main basic concepts, methods and results of the already very rich body of knowledge obtained recently at the attempts to understand, qualitatively and quantitatively, entanglement, laws and limitations of its distribution as well as its properties, structures and potentials for applications.

In order to demonstrate special power of quantum entanglement two puzzles "with science fiction overtones" are dealt with first. They look like exhibiting a telepathy.

⋆ Most of the paper has been written during the first author stay with the ERATO project. Support of the grants GAČR 201/01/0413, CEZ:J07/98:143300001 is also to be acknowledged.

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 25–68, 2001.

1 Introduction

Quantum physics has been well known for a long time for a variety of strange, puzzling, mysterious and even paradoxical phenomena that withstood numerous attempts to be dealt with by some of the brightest people in science. As a consequence, most of the (quantum) physicists got used to live with the understanding that quantum physics is a superb theory all quantum physicists (should) know how to use, but actually nobody understands it fully. Puzzling, mysterious and paradoxical quantum phenomena actually did not seem to disturb most of the working (quantum) physicists too much. There was no conflict discovered between predictions of theory and experiments. Only philosophers of science, working on different interpretations, could come from time to time with some interpretations, with which usually only very few others fully agreed. The situation is quite well characterized by the observation that philosophers of science can hardly agree even on what the term "interpretation of quantum physics" should mean.

Quantum theory performs its prediction role superbly, but lags behind concerning its other very basic task. Namely, to find explanations for all fundamental quantum phenomena; why they happen, how they happen, and how much they cost in terms of various physical resources. Quantum entanglement and quantum measurement are perhaps the best examples of such phenomena about which we still know far too little.

Quantum entanglement is a subtle correlation between subsystems of a quantum system. A feature of the system that cannot be created by local operations that act on subsystems only, or by means of classical communications.

Quantum entanglement is arguably the most specific of all inherently quantum phenomena. In spite of the fact that quantum entanglement and its nonlocality implications were basically known for more than 65 years, quantum entanglement started to be of an intensive interest only after it has turned out that it can be useful for (quantum) teleportation, to assist in speeding up quantum information processing, to enhance capacity of quantum channels, to realize unconditionally secure cryptographic key generation and to make quantum communication in some cases much more efficient. Currently, it is believed that a better comprehension of quantum entanglement can have important impact also on our overall understanding of key quantum phenomena (and that it can lead to the view of the nature of quantum mechanics from a different angle as till now). It is also expected that practical utilization of quantum entanglement can have a broad impact on the progress in making use of quantum resources for various areas of science and technology.

A better understanding of quantum entanglement, of ways it is characterized, created, detected, stored, manipulated, transformed (from one form to another), transmitted (even teleported) and consumed (to do some useful work), as well as of various types and measures of entanglement, is theoretically perhaps the most basic task of the current QIP research. In short, quantitative and qualitative theory of the entanglement is much needed.

It is interesting also to observe that the whole field of quantum information theory, and especially quantum entanglement, is *theory driven*. It did not arise because of observations from some experiments or from very specific needs of some applications.

Historically, quantum entanglement got a large attention in 1935 in connection with the so called EPR-paper (Einstein et al., 1935), and the related paper by Schrödinger (1935).[1] The EPR-paper concentrated on nonlocality quantum entanglement manifests and consequently on the question of the relation between quantum theory concepts and physical reality. Schrödinger recognized profoundly non-classical correlation between the information which entangled states give us about the whole system and its subsystems.

After 1935 entanglement has been seen as a troublemaker, and the problem was to get rid of it. Nowadays, it is seen as a precious resource and in the study of quantum entanglement much dominate those problems that are related to the need to understand and utilize entanglement as an information processing and communication resource.

2 Puzzles

Two entertaining puzzles discussed next are to illustrate power of entanglement to assist in creation of a sort of telepathy illusion – a pseudo telepathy. The first one is due to Hardy (1998).

2.1 Stage Telepathy – Puzzling Behaviour of Alice and Bob

Alice and Bob are on a stage, very far from each other (so far that they cannot communicate), and they are simultaneously, but independently and randomly asked again and again either a "food question" or a "colour question".

- **FOOD question**: What is your favorite meal?
  ANSWER has to be either **carrot** or **peas**.

- **COLOUR question**: What is your favorite colour?
  ANSWER has to be either **green** or **red**.

The audience observes that their answers satisfy the following conditions:

- If both are asked colour-questions then in about 9% of the cases they answer **green**.
- If one of them is asked colour-question and answers **green** and the other is asked food-question then (s)he answers **peas**.
- If both are asked food-questions they never both answer **peas**.

[1] The term *entanglement* is a free translation of the German term "Verschränkung" introduced by Schrödinger.

It is not difficult to show that within the classical physics there is no way that Alice and Bob could invent a strategy for their answers before they went to stage in such a way that the above mentioned rules would be fulfilled. However, there is a quantum solution, and actually quite an easy one.

Let $|p\rangle$ and $|c\rangle$ be two arbitrary orthogonal states in H_2 and let

$$|r\rangle = a|p\rangle + b|c\rangle,$$
$$|g\rangle = b|p\rangle - a|c\rangle.$$

Let Alice and Bob at the very beginning, before they went to stage, create two particles in the state

$$|\psi\rangle = N(|r\rangle|r\rangle - a^2|p\rangle|p\rangle),$$

where N is a normalization factor, and later each of them takes his/her particle with him/her to the stage.

If any of them is asked the colour-question, then (s)he measures his/her particle with respect to the $\{|r\rangle, |g\rangle\}$-basis and answers in accordance with the result of measurement.

If any of them is asked the food-question (s)he measures his/her particle with respect to the $\{|p\rangle, |c\rangle\}$-basis and responds in accordance with the result of measurement.

It is a not difficult exercise to show that in this way Alice's and Bob's responses follow the rules described above (9% comes from an optimization in one case).

2.2 Guess My Number Puzzle

A very nice technical result concerning the communication power of entanglement, obtained by van Dam, Høyer and Tapp (1997), was attractively formulated into a puzzle, by Steane and van Dam (2000), as follows.

Alice, Bob and Charles are located in isolated booths and are to guess, each time the moderator distributes secretly among them a number of apples, in total at most 4, potentially dividing some of the apples first into halves and distributing also halves, whether the total number of distributed apples was even or odd. If they guess correctly, they win.

The guess is actually to be made by Alice and the only communication allowed between the contestants, Alice, Bob and Charles, is that each of Alice's companions is allowed to send one bit to Alice (after seeing the portion of apples he got).

Contestants are allowed, before going to isolated booths, to agree on a strategy and to take with them whatever they want. However, once they are in booths, they are not allowed to communicate in some other way than sending to Alice one bit at each round of game.

The basic question is whether it is possible that contestants always win? That Alice is always able to determine whether the number of apples distributed is even or odd?

There is no way for Alice, using only classical communications, to guess in more than 75% cases the correct answer. However, there is the following quantum solution.

- Before going to booths contestants create three particles in the GHZ state $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$ and each of them takes with him/her one of the particles.
- In the case Alice, Bob, and Charles get x_a, x_b or x_c apples, respectively, where $x_a + x_b + x_c$ is an integer smaller than 5, then each of them applies to his/her particle the rotation

$$|0\rangle\langle 0| + e^{ix_i\pi}|1\rangle\langle 1|.$$

  The resulting state is either

$$\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle), \quad \text{if } x_a + x_b + x_c \text{ is even,}$$

  or

$$\frac{1}{\sqrt{2}}(|000\rangle - |111\rangle), \quad \text{if } x_a + x_b + x_c \text{ is odd.}$$

- In order to enable Alice to determine which of the states they jointly own, each of the contestants applies on his/her qubit the Hadamard transformation, getting together either the superposition of the basis states of even parity

$$\frac{1}{2}(|000\rangle + |110\rangle + |101\rangle + |011\rangle),$$

  or a superposition of states of odd parity

$$\frac{1}{2}(|001\rangle + |010\rangle + |100\rangle + |111\rangle).$$

- Each of them then measures his/her qubit in the standard basis and Bob and Charles communicate the outcomes to Alice. They always win.

3 Quantum Entanglement and Its Nonlocality Implications

Quantum entanglement is a term used to denote inherently quantum and inherently global correlations between quantum subsystems of a composed quantum system. (The state of a composed classical system is composed of the states of its subsystems – this is not in general true for quantum systems). The existence of nonlocal quantum phenomena, and the puzzles and mysteries this leads to, are its main implications.

3.1 Basics about Entanglement

We first briefly overview the very basic concepts of quantum information theory related to quantum entanglement. For more concerning basic of QIPC see Gruska (1999, 2000).

A pure state $|\phi\rangle$ of a bipartite quantum system $A \otimes B$ is said to be **separable**, or *classically correlated*, if $|\phi\rangle = |\phi_A\rangle \otimes |\phi_B\rangle$, where $|\phi_A\rangle$ ($|\phi_B\rangle$) is a pure state of the quantum system A (B). A pure state of $A \otimes B$ is called unseparable or **entangled** (or quantumly correlated), if it is not separable – if it cannot be decomposed into the tensor product of pure states of A and B.

A mixed state[2] (density matrix) ρ of $A \otimes B$ is called **separable** if it can be written in the form $\rho = \sum_{i=1}^{k} p_i \rho_{A,i} \otimes \rho_{B,i}$, where all *weights* p_i are positive and ρ_{A_i} (ρ_{B_i}) is a density matrix of A (B).[3] A density matrix ρ is called unseparable, or **entangled**, if it is not separable. A density matrix is therefore entangled, if it is not a mixture of product states.

An important example of entangled states in $H_2 \otimes H_2$ are Bell states:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \qquad |\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle),$$

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle), \qquad |\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle).$$

The state $|\Psi^-\rangle$ is called **singlet**. It is often said that each of the Bell states contains one **ebit** of entanglement.

There are several reasons why to take the above definition of entanglement of density matrices as the most general one, even if it does not seem to be very natural (and not that a density matrix ρ is entangled if it cannot be written as a tensor product of some density matrices of subsystems A and B, what would seem to be a natural generalization of the pure state case).

- A (mixed) state is not useful for quantum teleportation, see page 41, if and only if it is separable in the above sense.
- Neither a change of basis nor local operations can create an entangled state from unentangled ones.
- For all natural measures of entanglement the amount of entanglement cannot be increased using local quantum operations and classical communications.

Understanding, characterization, classification, quantification as well as manipulation of multipartite entangled states is a very important and difficult problem, perhaps the main current challenge in the quantum information theory, that needs to be explored in order to get more insights into the possibilities of distributed quantum computing and networks. The problem is also of special

[2] Throughout this paper we often use term "mixed state" also for density matrices. The rational behind is that two mixed states are undistinguishable if their corresponding density matrices are the same.

[3] An equivalent definition is that $\rho = \sum_i p_i |\phi\rangle\langle\phi_i| \otimes |\psi_i\rangle\langle\psi_i|$, where $|\phi_i\rangle$ and $|\psi_i\rangle$ are pure states in two Hilbert spaces and $p_i > 0$.

interest outside of QIP, for physics in general. For example, for understanding of dynamical systems – especially many-body systems with strong fluctuations and for improving information gathering facilities of experimental physics.

Error correcting codes are one example of an important area where multipartite entanglement plays a crucial role and where quite a few insights have been obtained about it.

A (mixed) state $|\rho\rangle$ of a m-partite quantum system represented by the Hilbert space $H^{(1)} \otimes H^{(2)} \otimes \ldots \otimes H^{(m)}$ is called **separable** if it can ve written in the form

$$\rho = \sum_{i=1}^{n} p_i \rho_{i,1} \otimes \rho_{i,2} \otimes \ldots \oplus \rho_{i,m},$$

where each $\rho_{i,j}$ is a density matrix of $H^{(j)}$ and $p_i > 0$. Otherwise, $|\phi\rangle$ is called **unseparable** or **entangled**. (Such a state cannot be created by m parties if the i-th party works with a quantum system corresponding to the Hilbert space $H^{(i)}$ and uses only local quantum operations and classical communications.)

Entanglement of multipartite systems is theoretically a very complex issue and it has many degrees and forms, as discussed in Section 9. For example, if an m-partite state is entangled it can still be separable if some of the parties get together and can perform global operations on the composition of their quantum subsystems. For example, if $I_1, I_2, \ldots I_k$ is a partition of the set $\{1, 2, \ldots, m\}$, then we say that an m-partite state is $I_1 : I_2 : \ldots I_k$ separable, if it is separable on the Hilbert space $H^{(I_1)} \otimes H^{(I_2)} \otimes \ldots \otimes H^{(I_k)}$, where $H^{(I_s)}$ is a tensor product of Hilbert spaces $\{H^{(i)} \,|\, i \in I_s\}$.

For example, the so-called Smolin (2000) state

$$\rho_s = \frac{1}{4} \sum_{i=1}^{4} |\Phi_i\rangle\langle\Phi_i| \otimes |\Phi_i\rangle\langle\Phi_i|,$$

where $\{|\Phi_i\rangle \,|\, i \in \{1, 2, 3, 4\}\}$ are all Bell states, is entangled in the Hilbert space $A \otimes B \otimes C \otimes D$, where A, B, C, D are two dimensional Hilbert spaces, but it is $\{A, B\} : \{C, D\}$, $\{A, C\} : \{B, D\}$ and $\{A, D\} : \{B, C\}$ separable.

Example. The so-called **GHZ states** are states of the form $\frac{1}{\sqrt{2}}(|abc\rangle \pm |\bar{a}\bar{b}\bar{c}\rangle)$, where $a, b, c \in \{0, 1\}$ and at most one of a, b, c is one. They are neither $\{A, B\} : \{C\}$ nor $\{A\} : \{B, C\}$ nor $\{A, C\} : \{B\}$ separable, if three parties are denoted as A, B and C.

Special Mixed States. Several families of mixed states play important role in quantum information processing theory, especially the so called **Werner states**. They form a family of one parameter states

$$W_f = f|\Psi^-\rangle\langle\Psi^-| + \frac{1-f}{3}(|\Psi^+\rangle\langle\Psi^+| + |\Phi^+\rangle\langle\Phi^+| + |\Phi^-\rangle\langle\Phi^-|), \quad 0 \le f \le 1$$

A Werner state W_f (W_p) is entangled if and only if $f \ge \frac{1}{2}$.

3.2 Entanglement and Nonlocality

Quantum theory implies, and (not fully perfect yet) experiments confirm, that a set of particles can be in an entangled state even if they are much space separated. As a consequence, a measurement on one of the particles may uniquely determine the result of measurement on much space separated particles. Einstein called this phenomenon "spooky action at a distance", because measurement in one place seems to have an instantaneous effect at the other (very distant) place. In other words, measurements of entangled states may cause " (weakly) nonlocal effects". (The term "weakly" refers to the fact that such a nonlocality cannot be used to send messages.)

It is this "quantum nonlocality" that entanglement exhibits, that belongs to the most specific and controversial issues of the quantum world. A set of particles in an entangled state can therefore be seen as a special quantum channel through which outcomes of one measurement can have immediate impact at much distant places.

Nonlocality of the physical world is not a new issues. The existence of nonlocal phenomena has been assumed by Newton when he developed his theory of gravity. It has been later rejected by Einstein when he developed his theory of relativity.

Various ways to see "nonlocal effects" are closely related to fundamental questions concerning the nature of physical reality and causation.

The orthodox Copenhagen interpretation of quantum physics sees the basic reality behind quantum theory as "our knowledge". This is well encapsulated in Heisenberg's famous statement (see Stapp, 2000):

> *The conception of the objective reality of the elementary particles has thus evaporated not into the cloud of some obscure new reality concept, but into the transparent clarity of a mathematics that represents no longer the behaviour of the particles but rather our knowledge of this behaviour.*

Einstein can be seen as the main representative of a quite different view. Namely, that it is not acceptable that a major physical theory would not refer to the physical reality. Well known are also positions from the EPR-paper (Einstein et al., 1935), that a proper physical theory should have the following features:

Completeness: "In a complete theory there should be an element corresponding to each element of reality."

Locality: "The real factual situation of system A is independent of what is done with system B, which is spatially separated from the former."

Reality: "If, without in any way disturbing a system, we can predict with certainty (i.e., with probability equal to unity) the value of a physical quantity, then there exists an element of physical reality corresponding to this physical quantity."

In particular, they suggested that quantum states do not provide a complete description of the physical reality, and therefore there have to exist additional, the so called "hidden" variables, whose objective values would unambiguously determine the result of any quantum experiment.

3.3 Bell Theorem, Its Proofs and Experimental Verification

Einstein, and many others, believed that such mysteries as entanglement seems to imply are due to the fact that the current quantum theory is incomplete, and he believed that by using some "hidden variables" one could develop a complete theory without non-local influences, where one could believe in the existence of an objective reality for quantum theory concepts.

There have been several attempts to develop a hidden variable interpretation of quantum theory. Arguably the most successful has been one due to Bohm (1952). However, a simple, but ingenious, Gedanken experiment suggested by Bell (1964), led to the conviction, for many, that all attempts to develop a hidden variable theory of quantum physics have to fail.

Bell showed that ideas suggested in the EPR paper, about locality, reality and completeness, are incompatible with some quantum mechanical predictions concerning entangled particles. He showed, for a simple Gedanken experiment, that the average values of certain random variables related to the experiment have to satisfy certain inequalities provided quantum physics satisfies a hidden variable theory and that these average values do not satisfy the above mentioned inequalities provided the current quantum theory holds.

All Gedanken experiments and inequalities of the Bell type allow to test experimentally quantum nonlocality — such experiments are called Bell experiments. The first quite convincing test of Bell inequalities was performed by Aspect in 1982. Since then a variety of other experiments have been performed and they brought quite impressive confirmation of the current quantum theory and of its nonlocality. However, none of them have been without loopholes. The loopholes are mainly of two types. Either all particles on which measurements are performed are not too far or quality of detectors is not fully sufficient (see Percival, 1999). The views differ concerning our possibilities to perform a Bell experiment without loopholes (in other words to close all loopholes) in the near future.

All experiments performed so far make use of untestable supplementary assumptions. For example the "fair sampling assumption". Its basic idea is that "the detectors sample the ensemble in a fair way so that those events in which both particles are detected are representable of the whole ensemble". Or the assumption "that detector efficiency is not dependent on local "hidden variables"".

Without the above assumptions the detectors used in the experiments need to have some efficiency (estimated to be at least 67%) what is far more than what has been achieved so far.

It is also far from being completely clear whether we can performed Bell experiments with ever increasing precision and whether we do not discover some laws of Nature that prevents us from doing that.

In order to perform Bell experiments it is needed to generate pairs of entangled particles. The so called Mermin experiments and inequalities allow to test nonlocality using three and more particle entanglement. In these cases requirements on quality of detectors are smaller and quantum nonlocality can be determined more clearly. However, creation of entangled particles is a non-trivial

problem and its complexity grows rapidly with the number of particles that have to be mutually entangled.

One can say that Bell experiments, based on Bell states, are able to show that statistical predictions based on quantum hidden variable theories differ from those provided by quantum mechanics. Later, Greenberger et al. (1989) have demonstrated that Gedanken experiments based on GHZ states can show that also definite predictions of any hidden variable theory differ from those provided by quantum mechanics.

3.4 Creation of Entangled States

Creation of entangled states in labs is a key and non-trivial problem for a variety of experiments concerning entanglement and of QIPC in general.

There are various ways how to create entangled states. Let us list some of the main ones.

Experimental generation using special physical phenomena.
Entangled photons, atoms, ions and nuclear spins have been experimentally produced. Entangled atoms were produced using cavity quantum electrodynamics, entangled ions were produced in electromagnetic Pauli traps, controlled entanglement between nuclear spins within a single molecule was achieved using NMR technology. Several types of entanglement were produced: polarization entanglement, time-energy entanglement, position and momentum entanglement, ...

There are four basic methods how to create experimentally an entangled state: (a) a physical source is just producing entangled states; (b) A source may produce independent quantum states that are component of an entangled superposition and then an apparatus has to be used to shift them in time so resulting entangled state is obtained; (c) to project a state (entangled or not) into an entangled one — see entanglement swapping bellow.

Sources of entanglement in condense-matter physics are now being intensively explored.

Most of known mechanisms for generating entangled particles depend on the specific nature of the systems involved. However, also more general schemes for entangling particles start to emerge, see for example Bose and Home (2001). What is much needed are sources for which there is a high degree of control over the degree of entanglement and purity of states.

To generate perfect pure entangled states, especially in the case of more particles is a very demanding task. Sometimes therefore the so called pseudo-pure states are used as sufficient.

Application of unitary operations to separable states. For example, all Bell states can be obtained if XOR operation is applied to some separable states $|\phi\rangle|a\rangle$, where $a \in \{0, 1\}$ and $|\phi\rangle$ is a suitable one qubit state.

Actually, some laboratory techniques to design entangled two-qubit states are quite similar. Some physical process is applied to two separable qubits to make them interact. On a general level the potential of this way of designing

two-qubit entangled states have been investigated by Kraus and Cirac (2000). They asked and answered the following question. Given a two-qubit unitary transformation U, what is the maximum of entanglement it can be obtained if U is applied to two qubits. In other words, what is the maximum of entanglement U can create, and for which qubit states.

Entanglement swapping — teleportation of entanglement. It used to be thought that if some distant particles are in an entangled state, then some time ago they had to interact and only later they had been separated. However, it is now clear that no direct interaction is necessary in order to produce entanglement between distant quantum systems. For example, entanglement can be teleported. A so-called **entanglement swapping** protocol has been proposed by Zukovski et al. (1993) — and experimentally verified by Pan et al. (1998).

The basic idea is simple. If we have two pairs of particles entangled in a Bell state (P_1 with P_2 and P_3 with P_4), Figure 1a, and a Bell measurement on particles P_2 and P_3 is performed, then we get two new entangled pairs: P_1 with P_4 and P_2 with P_3, both in a Bell state.

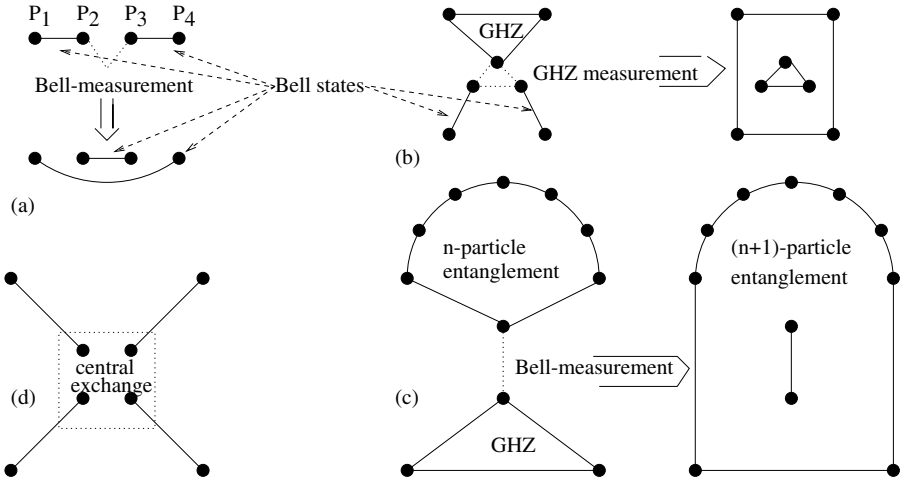


Fig. 1. Entanglement swapping and its generalizations

Technically, for any $a, b, c, d \in \{0, 1\}$, if a Bell operator is applied to the state

$$\frac{1}{2}(|abcd\rangle + |\bar{a}\bar{b}cd\rangle + |ab\bar{c}\bar{d}\rangle + |\bar{a}\bar{b}\bar{c}\bar{d}\rangle),$$

to its two middle qubits, then the resulting state is such that the particles P_1 and P_4, as well as P_2 with P_3, are in a Bell state. (Observe that entanglement swapping actually allows superluminal establishment of entanglement between two distant parties.)

It is easy to see that by creating several intermediate stations for producing Bell states and for performing Bell measurements one can significantly speed with entanglement swapping the task to supply very distant users with Bell states.

An elegant generalization of entanglement swapping to an arbitrary number q of quantum systems $\mathcal{S}_j$ composed from an arbitrary number m_j of qudits has been worked out by Bouda and Bužek (2000). Each of the systems $\mathcal{S}_j$ is supposed to be in a maximally entangled state of m_j qudits, while different systems are not correlated. They showed that when a set $\sum_{j=1}^{q} a_j$ of particles, with a_j particles from the system $\mathcal{S}_j$, are subjected to a generalized Bell measurement, the resulting set of particles collapses into a maximally entangled state. They also showed how to generalize results for the case of continuous variables.

Peres' Paradox. A curious modification of the entanglement swapping was developed by Peres (2000) using a simple protocol at which entanglement is produced a posteriori, after entangled particles have been measured and may no longer exists (!).

3.5 Local Quantum Operations and Classical Communication

In quantum information processing literature one often considers manipulations with multipartite, especially bipartite, systems at which only local operations and classical communications are allowed (by particular parties involved, each one in the possession of one of the subsystems). For this cases the following acronyms start to be used:

LOCC — local unitary operations and (two-way) classical communication.

LQCC — local quantum operations (any, including superoperators and POVM measurements[4]) and (two-way) classical communications.

SLQCC — stochastic local quantum operations and classical communication (two states are SLQCC interconvertible, if each of them can be obtained from the other one using LQCC with non-zero probability).

EALQCC — entanglement-assisted local quantum operations and classical communication (Jonathan and Plenio, 1999) — communicating parties are allowed, in addition to features offered by LQCC, first to "borrow" some entanglement, then to use it for local operations and classical communications, but without consuming it.

3.6 Nonlocality without Entanglement

Nonlocality is often identified with quantum entanglement. On one side, each pure entangled state violates a Bell type inequality and exhibits a nonlocality. However, a form of "nonlocality without entanglement" was demonstrated by

[4] Word of caution: often notation LOCC is used instead of LQCC

Bennett et al. (1998). They created an orthogonal set of product states of two qutrits (states in H_3) that cannot be reliably distinguished by LQCC. (They have shown that there is a gap between mutual information obtained by any joint measurement on these states and a measurement at which only local operations are performed.)

4 Detection of Entanglement

It is fairly easy to determine whether a given pure bipartite state is entangled. On the other hand, to determine whether a given density matrix is entangled seems to be a very nontrivial problem. Non-triviality of the problem is due to the fact that in the density matrices quantum correlations are weak and a manifestation of quantum entanglement can be very subtle (see Horodeckis, 2001).

4.1 Separability Criteria

By a **separability criterion** we call a condition for density matrices that is satisfied by all separable density matrices and it is not satisfied by all density matrices. More of entangled density matrices do not satisfy a separability criterion, stronger this criterion is. For example, all Bell inequalities (that are always satisfied by separable states) represent an interesting set of (not very strong) separability criteria.[5]For each pure entangled state there is therefore a set of measurements that allows to detect its entanglement.

A simple, but powerful, separability criterion, the so called **positive partial transposition criterion** (PPT-criterion) is due to Peres (1996). It is based on the observation that if a density matrix ρ of a composed (bipartite) quantum system $A \otimes B$ is separable, then its *partial transpose* $\bar{\rho}$ has to have positive eigenvalues.

A partial transpose $\bar{\rho}$ of ρ is obtained from ρ by making a transposition in ρ with respect to only one of the subsystems. Formally, if both A and B are n-dimensional Hilbert space and a density matrix ρ of $A \otimes B$ is decomposed into $n \times n$ submatrices, then the transposition of ρ, is obtained just by making transposition of submatrices as follows:

$$\rho = \begin{pmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \dots & A_{nn} \end{pmatrix}, \quad \bar{\rho} = \begin{pmatrix} A_{11}^T & \dots & A_{1n}^T \\ \vdots & \ddots & \vdots \\ A_{n1}^T & \dots & A_{nn}^T \end{pmatrix}$$

[5] The so called Bell-CHSH inequalities are inequalities of the form $Tr(\rho B) \leq 2$, where B are matrices of the form $B = \bar{r}_1\sigma \otimes (\bar{r}_2 + \bar{r}_3)\sigma + \bar{r}_4\sigma \otimes (\bar{r}_2 - \bar{r}_3)\sigma$, where $\bar{r}$'s are unit real vectors and if $\bar{r} = (r_1, r_2, r_3)$, then $\bar{r}\sigma = r_1\sigma_x + r_2\sigma_y + r_3\sigma_z$, where $\sigma_x, \sigma_y, \sigma_z$ are Pauli matrices.

For example, the density matrix ρ of the Bell state Ψ^+ and its transpose $\bar{\rho}$ have the form

$$\rho = \frac{1}{2}\begin{pmatrix} 0&0&0&0 \\ 0&1&1&0 \\ 0&1&1&0 \\ 0&0&0&0 \end{pmatrix}, \quad \bar{\rho} = \frac{1}{2}\begin{pmatrix} 0&0&0&1 \\ 0&1&0&0 \\ 0&0&1&0 \\ 1&0&0&0 \end{pmatrix},$$

and $\bar{\rho}$ has $-\frac{1}{2}$ as one of the eigenvalues.

A state ρ is called **PPT**-state (**NPT-state**) if its partial transpose is positive (is not positive). Peres' criterion was shown by Horodeckis (1998c) to be a necessary and sufficient condition for density matrices of bipartite systems of dimension 2×2 and 2×3 to be entangled.

4.2 Entanglement Witnesses

A very interesting and useful approach to the detection of entanglement is based on the concept of **entanglement witness** introduced by Horodeckis (1996) and coined by Terhal.

An entanglement witness for density matrices of a Hilbert space H of dimension n is any matrix (observable) W of dimension n such that

- $Tr(W\rho) \geq 0$ if $\rho \in H$ is a separable density matrix.
- There is an entangled density matrix $\rho \in H$ such that $Tr(W\rho) < 0$.

In other words an entanglement witness can detect entanglement in density matrices.

Swapping (or flipping) operator V defined by $V(|\phi\rangle \otimes |\psi\rangle) = |\psi\rangle \otimes |\phi\rangle$ is an example of an entanglement witness.

Since an entanglement witness can fail to discover entanglement a natural question is to find as best as possible entanglement witnesses. Namely, such that do not fail at all or fail very rarely to detect entanglement.

An entanglement witness W is called **optimal**, see Lewenstein et al. (2000), if there is no other entanglement witness that detects the same entanglement as W and more. Lewenstein et al. (2000) also found a way to design optimal witnesses. This allows well experimentally detect quantum entanglement. For example, the optimal entanglement witness for the state $\cos\theta|00\rangle + \sin\theta|11\rangle$ has the form

$$W = \frac{1}{2}(|01\rangle\langle 01| + |10\rangle\langle 10| - |00\rangle\langle 11| - |11\rangle\langle 00|).$$

5 Power of Entanglement

The role of entanglement as of an important computational and communicational resource is often much emphasized. There have been many strong statements about the key role of entanglement for having quantum information processing to be more powerful than classical one. However, much more research seems to be needed to get a really full (or at lest sufficient) understanding of the role of entanglement for communication and, especially, for computation.

5.1 Power of Entanglement for Computation

Gottesman and Chuang (1999) have shown that entanglement is a computational primitive because it is possible to realize any quantum computation by starting with some GHZ states and then performing one qubit operations and Bell measurements.

There are also other results showing that entanglement is a sort of substrate on which quantum computation can be performed. For example, Rausschendorf and Briegel (2000) have shown that universal quantum computation is possible by starting with a special multipartite entangled states and then performing only single qubit measurements, to model a circuit computation.

In addition, Shor's algorithms make heavy use of entanglement, and no way seems to be known how to perform efficient factorization, or to solve efficiently the hidden subgroup problem, without entanglement.

In spite of that it is far from clear that entanglement is the key resource behind the efficiency of fast quantum algorithms. Quantum superposition, parallelism and, especially, quantum measurement are other, perhaps even more important, quantum resources for computation.

It is also interesting to notice that quantum entanglement does not play a role in the current theory of quantum automata (quantum finite automata, quantum Turing machines, quantum cellular automata). It does not seem also that entanglement play a role at the definition of such quantum complexity classes as **BQP**.

To illuminate the role the entanglement has for efficient quantum computation is one of the key tasks of the current QIPC research. Unfortunately, it does not seem to be a very visible way how to do it.

5.2 Power of Entanglement for Communication – Quantum Communication Mysteries

While in quantum computation we merely believe that quantum mechanics allows exponential speed-up for some computational tasks, in quantum communication we can prove that quantum tools can bring even exponential savings.

Let us start our discussion of quantum communication by pointing out that there are actually good reasons to believe that quantum communication cannot be of special use. Indeed, Holevo theorem says that no more than n bits of classical information can be communicated by transmitting n qubits – unless two communicating parties are entangled and in such a case at most twice as many classical bits can be communicated, using quantum dense coding, see Section 6.

In addition, entanglement itself cannot be used to signal information — otherwise faster than light communication would be possible. Our intuition therefore says that there should be negative answers to the following fundamental questions (Brassard, 2001):

– Can entangled parties make better use of classical communication channels than non-entangled parties?

– Can entangled parties benefit from their entanglement even if they are not allowed any form of direct (classical or quantum) communication?

Surprisingly, as we are going to discuss now, and as the puzzles already indicated, our intuition in these cases has turned out to be very wrong.

5.3 Entanglement Enhanced Quantum Communication Complexity

In the so called *entanglement enhanced quantum communication model*, only classical bits are communicated, but communication is facilitated by an a priori distribution of entangled qubits among the communicating parties. Communication protocols are defined in the quantum case in a similar way as in the classical case.

An exponential gap between the exact classical and quantum entanglement enhanced communication complexity has been shown by Buhrman et al. (1998), for the exact solution of the following promise problem.

Let $f, g : \{0,1\}^n \to \{0,1\}$, and let $\Delta(f,g)$ be the Hamming distance between f and g which equals the Hamming distance of 2^n-bit strings $f(0)f(1)\ldots f(2^n-1)$ and $g(0)g(1)...g(2^n-1)$. Let EQ' be the partial function defined by

$$\mathrm{EQ}'(f,g) = \begin{cases} 1, \text{ if } \Delta(f,g) = 0, \\ 0, \text{ if } \Delta(f,g) = 2^{n-1}; \end{cases}$$

and undefined for other arguments. (This can be seen also as the requirement to solve the above equality problem under the promise that $\Delta(f,g) \in \{0, 2^{n-1}\}$.)

Raz (1999) presents another example of a communication promise problem, to compute a partial function, for which there is a quantum communication protocol with exponentially smaller communication complexity than any probabilistic communication protocol exhibits.

Open problem. Can we show an exponential gap between the classical communication complexity and the entanglement enhanced quantum communication complexity also for total functions in a non-promise setting?

5.4 Spooky Quantum Communication Complexity

The second question formulated on page 40 has also positive answer. It is the question whether entanglement can be used instead of communication. In other words, whether quantum pseudo telepathy is possible and what we can say about it quantitatively.

One way to approach the problem (Brassard, 2001), could seem to be to ask whether we can have a function $f : X \times Y \to Z$ such that it is not possible to determine $f(x,y)$ from the knowledge of either x or y, but two entangled parties can compute $f(x,y)$, if one is given an x and other a y. However, this cannot be the case, because this would allow faster than light communication. However, there is a way to find a positive answer to the above problem when computation of relations and not of functions is considered. *Spooky communication complexity*, Brassard et al. (1999), has been introduced as follows.

Let X, Y, A and B be sets and $R \subset X \times Y \times A \times B$. In the initiation phase, two later physically separated and entangled parties, Alice and Bob, are allowed to establish a strategy. After they are separated one of them is given an $x \in X$ and second a $y \in Y$ and their goal is to produce $a \in A$ and $b \in B$, each of them one, such that $(x, y, a, b) \in R$. It is said that *spooky communication* or *pseudo-telepathy* takes place, if the above task cannot be done for sure classically, but it can be done once these two parties share entanglement. **Spooky communication complexity** of the task is then defined as the amount of entanglement required to perform the task.

Brassard et al. (1999) has shown, for the relation R defined bellow, that the number of bits to compute the relation classically is exponentially larger than the number of entangled pairs shared by the parties at spooky communication (and therefore that spooky communication can be exponentially more effective).

Let k be an integer, $n = 2^k$, $X = Y = \{0,1\}^n$, $A = B = \{0,1\}^k$ and

$$(x, y, a, b) \in R \leftrightarrow \begin{cases} x = y \text{ and } a = b; \\ \Delta(x, y) = \frac{n}{2} \text{ and } a \neq b; \\ \Delta(x, y) \notin \{0, n/2\}. \end{cases},$$

where again $\Delta(x, y)$ denotes the Hamming distance. (In other words, if Alice and Bob are promised that their inputs are either the same or differ in half of bits and they have to produce the same outputs only if their inputs are the same.)

It has been shown that there is a constant c such that the above relation cannot be established classically with fewer than $c2^k$ bits provided k is sufficiently large, but k entangled pairs are sufficient for a spooky communication solution.

6 Quantum Teleportation and Superdense Coding

Perhaps the most interesting and inspiring demonstration of entanglement, at least so far, as a communication resource is quantum teleportation, due to Bennett et al. (1993), that allows transmission of an unknown quantum state to a distant place in spite of the impossibility of measuring or broadcasting states to be transmitted. Quantum teleportation is arguably one of the main achievements of quantum information theory.[6]

Experimental verification of quantum teleportation is still one of the main experimental challenges in quantum information processing. The first attempt to perform such a teleportation was successfully, to a certain degree, done by

[6] The so called *No-teleportation theorem*, one of the fundamental laws of quantum mechanics, says that *(classical) teleportation* is impossible. This means that there is no way to use classical channel to transmit faithfully quantum information. In more technical words there is no possibility to measure a quantum state in such a way that the measuring results would be sufficient faithful to reconstruct the state. (It is easy to see that classical teleportation would imply that quantum cloning is possible and this would imply that joint measurements are always possible and finally this would imply, through so called *Bell telephone*, see Werner (2000) for a lucid explanation, that superluminal communication would be possible.

Zeilinger's group in Innsbruck (see Bouwmeester et al. (1997)). Currently the most perfect teleportation experiment seems to be reported by Kim et al. (2000), as the first experiment that satisfies the following basic requirements: (a) arbitrary state can be teleported; (b) there is output state that is "instantaneous copy" of the to-be-teleported state; (c) all four Bell states can be distinguished in the Bell measurement.

6.1 Basic Teleportation Schemes

The most basic task is teleportation of pure quantum states, especially qubits.

Teleportation of Qubits. The basic teleportation protocol runs as follows. Alice and Bob share two particles in a Bell state $|\Phi^+\rangle$ (forming the so called *teleportation channel* Φ), and Alice posses a particle C in an unknown state $\alpha|0\rangle + \beta|1\rangle$. If Alice performs the Bell measurement on her particle and on the particle C, then Bob's particle gets with the same probability into one of the four states $\alpha|0\rangle + \beta|1\rangle$, $\beta|0\rangle + \alpha|1\rangle$, $\alpha|0\rangle - \beta|1\rangle$, $-\beta|0\rangle + \alpha|1\rangle$, and the classical result of the measurement, obtained by Alice, tells uniquely which of these four cases has materialized. Once Alice sends this classical information to Bob, he knows which of the Pauli matrices to apply to his particle to get it into the (unknown) state $\alpha|0\rangle + \beta|1\rangle$.

Remark. The following facts concerning quantum teleportation are worth to observe: (a) There is no need to know the state in order to be able to teleport it; (b) All quantum operations performed during teleportation are local; (c) All operations performed during teleportation are independent of the state being teleported; (d) Quantum teleportation can be seen as a "real teleportation" at which an object is at first dematerialized, then transfered and finally reassembled, at a different place — Bob's particle at the end of the teleportation is undistinguishable from Alice's particle at the beginning of teleportation; (e) In spite of the fact that maximally entangled state that Alice and Bob share is publicly known, teleportation represents an absolutely secure way of transferring quantum information at which no eavesdropping is possible — because no particle is transfered; (f) Observe that after the measurement Bob's particle is in the totally mixed state $\frac{1}{2}I$ and therefore a potential eavesdropper cannot learn anything about state being teleported this way.

6.2 Generalized Teleportation Schemes

There are, as one would expect, several generalization of the basic teleportation scheme to explore. One of them is to consider two party teleportation, but of more dimensional states than qubits. The other natural possibility is to consider multipartite teleportation

A general approach to two-party quantum teleportation has been worked out by Accardi and Ohya (1999) and especially by Albeverio and Fei (2000). The problem that has been solved is the following one: Given a state $|\phi_1\rangle$ of a Hilbert

space H_1, and an entangled state $|\psi_{2,3}\rangle$ of the tensor product of Hilbert spaces $H_2 \otimes H_3$, with all three Hilbert spaces having finite dimension and with dimension of H_1 being not smaller than that of H_3, conditions are established for a unitary transformation U on $H_1 \otimes H_2$ such that if on the state $U \otimes I(|\phi_1\rangle|\psi_{2,3}\rangle)$ a proper measurement is performed than the resulting state $|\psi_3\rangle$ of H_3 is such that the result of the above measurement uniquely determines a unitary transformation, not dependent on $|\phi_1\rangle$, which, when applied on $|\psi_3\rangle$, yields the state $|\phi_1\rangle$.

Concerning multipartite teleportation, Gorbachev and Trubilko (1999) at first showed a simultaneous teleportation of an unknown state $\alpha|00\rangle + \beta|11\rangle$ through a GHZ-channel to two parties. However, Marinatto and Weber (2000) showed that it is not possible to perform teleportation of any two-qubit state. Gorbachev et al. (2000) then showed how to teleport any entangled two qubit state using GHZ channel and three bits of classical information.

6.3 Remote State Preparation (RSP)

Let us now investigate the case that the party in need of teleporting a state has a full classical knowledge of the state. Is it reasonable to expect that in such a case either less ebits or less bits are needed for teleportation? At least in some cases?

The answer is, quite surprisingly, positive. As shown by Lo (1999) and Bennett et al. (2001), in such a case there is a non-trivial tradeoff between the ebits and bits. Bennett et al. (2001) showed, for example, that asymptotically classical communication cost of RSP, in the presence of a large amount of preshared entanglement, can be reduced to one bit for qubit provided four ebits are used (and becomes even less when transmitting part of a known entangled state). During RSP most of the preshared entanglement is not consumed and it can be recovered, using a backward communication, from Bob to Alice. On the other hand, if a large number of bits is used, say n, the number of ebits needed grows like $n2^{-n}$. Protocols developed by Bennett et al. (2001) work even for finite number of (identical) states Alice wants to prepare for Bob. A certain catch is that no protocol is known that creates states always faithfully and uses in all cases less classical communication, per qubit, than quantum teleportation.

6.4 Teleportation as a Computation Primitive

Teleportation illustrates well the known fact that quantum measurements can be seen as an important quantum computational primitives and that in some cases quantum measurement does not have to destroy quantum information irreversibly. Just the opposite, it can help to transfer quantum information from one place to another. In addition, as shown by Gottesman and Chuang (1999), generalized quantum teleportation can be seen as a universal quantum computational primitive.

6.5 Teleportation and Nonlocality

It seems intuitively clear that quantum teleportation is due to nonlocality that (measurements of) entangled states exhibit. However, quite surprisingly, Hardy (1999) has shown that concepts of teleportation and nonlocality are conceptually independent. Namely, he constructed a toy model which is local and within which no-cloning theorem holds, but still teleportation holds.

6.6 Dense Coding

Teleportation uses two bits to teleport one qubit. Dense coding (Bennett et al. (1993)), is dual to teleportation. It uses one qubit to send two bits.

Let Alice and Bob share the Bell state $|\Phi^+\rangle$. Depending on two bits to be send Alice performs on her particle one of the Pauli operators and sends her particle to Bob. Bob disentangles particles using XOR operator and measures his particle in the standard basis and Alice's particle in dual basis. The result of the measurement uniquely determines two bits Alice intended to transmit to Bob.

6.7 All Teleportation and Dense Coding Schemes

Werner (2000) brought an important new insight on possible teleportation and dense coding protocols. He showed not only that there is one-to-one correspondence between (optimal) teleportation and dense coding schemes (realized with minimal resources (with respect to Hilbert space dimension and classical information)), but that there is also a one-to-one correspondence between such schemes and schemes for the three other tasks: (a) construction of orthonormal bases of maximally entangled states; (b) construction of orthonormal basis of unitary operators with respect to Hilbert-Schmidt scalar product; (c) depolarizing operations whose Kraus operators can be chosen to be unitary.

On one side it is not surprising that there is a close relation between teleportation and dense coding protocols because actually they often perform basically the same operations only in the opposite order. One-to-one relations to other problems is more surprising. However, it helps to see complexity of the problem to design teleportation schemes because, for example, the problem of constructing orthonormal basis of unitary operators is shown to involve such not well understood tools as Hadamard matrices and Latin Squares.

A natural generalization is to consider dense coding over n-partite channels. As shown by Gorbachev et al. (2000), in such a case the improvement of the classical capacity of quantum channel is only $\frac{n}{n-1}$. Moreover, in this multipartite channel setting there are no such deep relations between teleporting and dense coding as discussed above.

6.8 Teleportation, Mixedeness and Dense Coding

A very natural question is how good are mixed states for teleportation and dense coding. One reason to ask this is that pure entangled states are harder to

produce than mixed states. Second reason is that mixed states can have more entanglement than pure separated states, if von Neumann entropy is taken as a good measure of mixedeness.

Horodeckis (1999) have shown that in the case of bipartite systems of dimension n the singlet fraction $F(\rho) < \frac{1}{n}$ implies that teleportation with ρ cannot be done with better than classical fidelity. Bose and Vedral (1999) improved that condition to $S(\rho) > \lg n + (1 - \frac{1}{n}) \lg(n+1)$. This means that for quantum systems of very large dimension, even an entropy close to maximal is not sufficient to ensure failure of teleportation.

Concerning dense coding, an entangled mixed state ρ fails to be useful for dense coding when $S(\rho) > \lg n$. Since the threshold $\lg n$ is smaller than that for teleporation, $\lg n + (1 - \frac{1}{n}) \lg(n+1)$, we can conclude that teleportation is "more robust" with respect to the external noise than dense coding.

For a general case that two parties A and B share mixed state ρ over $n \times n$-space the problem to determine capacity $C(\rho)$ of superdense coding channel of ρ is almost solved. Hiroshi (2000) established the bound $E_r(\rho) \leq C(\rho) \leq \lg n + E_r(\rho)$. Bowen (2001) conjectures that $C(\rho) = S(\rho_B) - S(\rho) + \lg 3$ and shows this for $n \leq 3$.

7 Entanglement Distillation – Concentration and Purification

Various applications (of entanglement) require that communicating or cooperating parties share maximally entangled pure states. For example, they can ensure perfect security of communication or rapid performance of certain forms of distributed computing. On the other hand, communicating and/or cooperating distant parties have to their disposal mostly not maximally entangled states and mainly, not pure, but mixed states. The question is then how to get out of them, using only LQCC, maximally entangled pure states. In this connection of large importance are two tasks.

- Entanglement concentration. How to create, using only LQCC maximally entangled pure states from not maximally entangled ones.
- Entanglement purification. How to distill pure maximally entangled states out of mixed entangled states.

Remark. Often term *distillation* is used to mean both concentration or purification.

7.1 Entanglement Concentration

Schmidt projection method (Bennett, 1996a) for entanglement concentration is asymptotically optimal, but requires to have (infinitely) many copies of the state.

Bennett et al. (1995) developed a method to make entanglement concentration starting with the state $(\alpha|00\rangle + \beta|11\rangle)^{\otimes n}$ and producing the expected

number of maximally entangled state equal to $H(|\alpha|^2) - \mathcal{O}(\lg(n))$. A simplification of the above method, with the same asymptotic performance, but also with detailed concentration circuits is due to Kaye and Mosca (2001).

A different and practically important task is to perform optimal entanglement concentration for a single pure bipartite state.

Jonathan and Plenio (1999) found a way, using Nielsen's (1998) theorem on transformation of bipartite states, to solve the following entanglement concentration problem.

Alice and Bob share a pure bipartite state $|\psi\rangle$ with Schmidt decomposition

$$|\psi\rangle = \sum_{i=1}^{n} \sqrt{\alpha_i}|\phi_i\rangle|\psi_i\rangle,$$

whose entanglement they want to concentrate using LQCC. Consider the set of local transformations that generate the maximally entangled state $|\theta_j\rangle = \frac{1}{\sqrt{j}} \sum_{i=1}^{j} |\phi_i\rangle|\psi_i\rangle$, $1 \leq j \leq n$ with probability p_j. Since von Neumann reduced entropy of $|\theta_j\rangle$ is $\ln j$, for the average amount of distilled entanglement it holds $\langle E\rangle = \sum_{j=1}^{n} p_j \ln j$. The following theorem shows how this quantity can be maximized.

Theorem The optimal entanglement concentration procedure for a bipartite state $|\psi\rangle$ with $\alpha_1 \geq \alpha_2 \geq \ldots \geq \alpha_n > 0$ is one that produces the maximally entangled state $|\phi_j\rangle$ with probability $p_j = j(\alpha_j - \alpha_{j+1})$. The corresponding optimal average entanglement is then

$$\langle E\rangle_{max} = \sum_{j=1}^{n} (\alpha_j - \alpha_{j+1}) j \ln j.$$

7.2 Entanglement Purification

The goal of entanglement purification techniques is to distill, from a set of m identical mixed two-qubit states, a set of $n < m$ maximally entangled pure states, say singlets, with the ratio $\frac{n}{m}$ as large as possible.

Global Purification Techniques. Conceptually simple purification technique was developed by Bennett et al. (1996). To describe this technique it is useful to consider two quantities on mixed states ρ in a Hilbert space H:

- **Singlet fraction** $F(\rho) = \langle \Psi^- | \rho | \Psi^- \rangle$;
- **Fully entangled fraction** $\mathcal{F}(\rho) = \max_\psi \langle |\psi|\rho|\psi\rangle$, where maximum is taken through all maximally entangled states.

Let Alice and Bob share n mixed states ρ with $F(\rho) > \frac{1}{2}$. They iterate the following steps:

1. They choose two pairs of particles, each pair in the state ρ, and apply to each of them unitary transformation $U \otimes U^*$, where U is randomly chosen. (This way states ρ are transformed to new identical states with the same singlet fraction.)
2. Each party performs XOR operation on their particles.
3. The pair of the target particles is measured in the standard basis and it is discarded. If the results agree the source part is kept and has a greater singlet fraction as ρ.

This method allows to get mixed states with arbitrarily large singlet fractions, but its asymptotic rate is zero and it requires that $F(\rho) > \frac{1}{2}$. However, a modification of the method, developed by Horodeckis (1997a), works for all two-qubit mixed states.

There is one crucial problem with the above (global) purification protocols. They rely heavily on carrying out collective measurements on a large number of shared states, assume infinite supply of shared states and work only asymptotically.

Local Purification Protocols. A practically important and theoretically interesting problem is then to determine how much purification, and when, can be done if we have only one mixed state to purify and only local quantum operations can be used. In other words, what is the power of local purification schemes?

It is known that some mixed states can be purified using local purification techniques. However, as shown by Kent (1998), no local purification protocol can produce maximally entangled pure state from any entangled mixed state. (For example, some Werner states cannot be purified locally.) In addition, Kent et al. (1999) designed a purification (and concentration) protocol for mixed states of two qubits that is optimal in the sense that protocol produces, with non-zero probability, a state of maximal possible entanglement of formation.

Practical Methods of Purification. Since distribution of entangled states between distant parties seems to be of large importance for various quantum communication protocols, such as quantum cryptography, quantum teleportation and so on, and entanglement purification is the main way to distill highly entangled states from less entangled states or mixed states, it is of large interest to find practical entanglement purification techniques.

The techniques discussed above are based on the use of XOR or similar quantum gates — on tools that are far from easy to implement perfectly enough to be really useful for purification. An important challenge of QIPC is therefore to look for other, more practical, purification techniques.

One such a technique has been developed by Pan et al. (2000). This techniques requires to use only simple linear optical elements that have very high precision. It should lead to a procedure that is within the reach of the current technology.

8 Entanglement Measures

Once it has been realized that quantum entanglement is a precious computation and communication resource, it has become important to quantify entanglement in order to determine how effectively we can process and communicate information.

It has turned out quite easy to quantify entanglement of pure states. On the other side, it has turned out more difficult to quantify entanglement of bipartite mixed states in a physically meaningful way, but quite a progress has been achieved. On the other hand, too little is known about the key problem in this area — how to quantify entanglement of multipartite mixed states.

8.1 Axioms

An entanglement measure is a real-valued function defined on mixed states (density operators) of bipartite or multipartite quantum systems satisfying some physically (and mathematically) motivated conditions.

The problem of finding minimal general conditions, usually termed "axioms" in physics literature, each good measures of entanglement should satisfy, has been dealt with for quite a while. There are several, more or less compatible, proposals for such axioms. For example, in Bennett et al. (1996a): (a) $E(\rho) = 0$ iff ρ is separable; (b) invariance under local unitary operations; (c) the expectation of entanglement should not increase under LQCC; (d) additivity: $E(|\phi\rangle \otimes |\psi\rangle) = E(|\phi\rangle) + E(|\psi\rangle)$, for the case that $|\phi\rangle$ and $|\psi\rangle$ are independent bipartite states; (e) stability with respect to a transfer of a subsystem from one party to another. (In any tripartite state $|\psi\rangle$ of a system $A \otimes B \otimes C$ the bipartite entanglement of $A \otimes B$ with C should differ from that of A with $B \otimes C$ by at most the entropy of B.)

Horodecki et al. (1999) formulated perhaps so far the most elaborated set of axioms a measure of bipartite entanglement should satisfy. Their axioms seem well to reflect current physical understanding of the subject. (a) non-negativity, $E(\rho) \geq 0$; (b) $E(\rho) = 0$ if ρ is separable; (c) normalization: $E(|\Phi^+\rangle\langle\Phi^+|) = 1$; (d) monotonicity under local operations — if either of the parties sharing a pair of particles in the state ρ performs an operation leading to a state ρ_i with probability p_i, then $E(\rho) \geq \sum_i p_i E(\rho_i)$; (e) convexity: $E(\sum_i p_i \rho_i) \leq \sum_i p_i E(\rho_i)$; (f) partial additivity: $E(\rho^{\otimes n}) = nE(\rho)$; (f) continuity: if $\lim_{n\to\infty} \langle \Phi^{+\otimes n} | \rho_n | \Phi^{+\otimes n} \rangle = 0$, then also $\lim_{n\to\infty} \frac{1}{n}[E(\Phi^{+\otimes n}) - E(\rho_n)] = 0$, where ρ_n is some joint state of n pairs.

The underlying thesis behind the above set of axioms is that the entanglement of distillation is a good measure of entanglement (and therefore there is no "iff" in the axiom (b) — because the bound entanglement, discussed in Section 11, is not distillable).

It has been claimed that for pure states von Neumann reduced entropy is the only measure that satisfies relevant axioms from the above set of axioms. However, as discussed also by Rudolph (2000), who analyzes current state of art

in this area, the problem of proper axioms seems still to require a more formal study, and probably some other axiom(s) may need to be added.

The case of measures of entanglement for multipartite systems is widely open.

8.2 Entanglement of Pure States - Entropy of Entanglement

For a pure state $|\phi\rangle$ of a bipartite system $A \otimes B$ there seems to be a single good measure of entanglement $E(|\phi\rangle)$, defined using relative von Neumann entropy and $\rho = |\phi\rangle\langle\phi|$ as follows.

$$E(|\phi\rangle) = -Tr(\rho_A \lg \rho_A) = -Tr(\rho_B \lg \rho_B).$$

Hence, entanglement of a pair of quantum states is defined as the entropy of either member of the pair.

This measure satisfies all axioms defined above. Moreover, any set of bipartite pure states the total sum of entanglement of which is an E can be reversibly interconverted into any other set of bipartite pure states with the same total entanglement E.

8.3 Measures of Entanglement of Mixed Bipartite States and Their Properties

Qualitatively, a bipartite density matrix ρ is entangled if some in principle physically possible transformations on ρ cannot be realized if both parties perform only local quantum operations and classical communication.

In spite of the large research effort, we do not have yet good quantitative understanding of the entanglement of mixed states. Let us list some of the basic questions for which we do not have yet satisfactory answer:

1. How to determine efficiently whether a density matrix is entangled? (Discussed in Section 4).
2. How to measure entanglement of bipartite mixed states?
3. Is entanglement irreversibly lost if we attempt to convert it from one mixed state to another?

One of the problems behind is that a density matrix can correspond to infinitely many mixtures of pure states and these pure states may have very different entanglement properties. For example, the completely mixed state $\frac{1}{2}I$ of two qubits corresponds to the equally probably mixture of (maximally entangled) Bell states, but also to a mixed state consisting of only unentangled states.

Currently, there seems to be no "canonical-way" to quantify entanglement of bipartite states and it seems that no single measure of entanglement of mixed states is proper in all situations/applications.

Entanglement of Formation E_f, (Bennett et al. 1996a), is defined for a density matrix ρ by

$$E_f(\rho) = \min\{\sum_i p_i E(\psi_i) \mid \rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|\},$$

where ρ is the density matrix for the mixed state $\bigoplus_{i=1}^{n}(p_i, \psi_i)$. In other words, entanglement of formation, $E_f(\rho)$, is the least expected entanglement of any ensemble of pure states with ρ as the density matrix.

Entanglement of formation is known exactly for many states, including all two-qubit states (see Hill and Wooters, 1997), having no more than two non-zero eigenvalues. In addition, Uhlmann (1997) has shown how to compute entanglement of formation for those states where E_f is not known to have a closed form.

An important variant of the entanglement of formation is its asymptotic version

$$E_f^{\infty}(\rho) = \lim_{n\to\infty} \frac{E_f(\rho^{\otimes n}}{n}),$$

even it seems to be very hard to compute.

Open problem. Is entanglement of formation additive? (A positive answer would imply $E_f = E_f^{\infty}$).

Entanglement of Distillation (distillable entanglement), E_d (see Bennett et al. 1996a), is intuitively, the maximum asymptotic yield of singlets that can be produced from the given mixed state by local operations and classical communication.

Distillable entanglement can be further classified by the type of classical communication allowed (no classical communication (E_{d0}), one-way classical communication (E_{d1}), and (E_{d2}) two-way classical communication) and depends also on the type of the local operations allowed.

Another way to define, informally, entanglement of distillation of a mixed state ρ is the maximum over all allowable protocols of the expected rate at which singlets can be obtained from a sequence of states ρ. For a rigorous treatment of the entanglement of distillation see Rains (1998).

One can also say that if Alice and Bob share n pairs of particles in mixed state ρ, then they can faithfully teleport $nE_d(q)$ qubits.

One of the fundamental questions concerning distillation of quantum states was whether distillation is in principle an irreversible process. More technically, whether $E_d(\rho) < E_f^{\infty}(\rho)$ for some states ρ. Horodeckis (1999a) have shown that this is indeed the case.

Entanglement of Assistance, E_a, (DiVincenzo et al. 1998), is in a sense dual to the entanglement of formation

$$E_a(\rho) = \max\{\sum_i p_i E(\psi_i) \,|\, \rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|\},$$

Entanglement of Relative Entropy. E_r, (Vedral and Plenio, 1997) defined by

$$E_r(\rho) = \min_{\rho' \in \mathcal{D}} (Tr\rho(\lg \rho - \lg \rho')),$$

where $\mathcal{D}$ denotes the set of unentangled mixed states.

In spite of the fact that this entropy has mainly mathematical motivation it has turned out as powerful tool to study entanglement.

Relations among Different Measures of Entanglement. The following relations hold among the above entanglement measures (see Vedral and Plenio, 1997, and Bennett et al. 1996a):

$$E_{d0} \leq E_{d1} \leq E_{d2} \leq E_r \leq E_f.$$

All these entanglement measures are reduced to the entropy of entanglement for pure states.

Remark. The development of quantitative theory of entanglement can be seen as a major task of QIPC. There is already a large body of (incomplete) knowledge about the above measures of entanglement. However, there are also good reasons to be cautious about their importance because so far there are not too many results connecting these measures of entanglement with other major problems of QIPC. A search for new measures of entanglement and new ways how to approach this problems are clearly much needed.

Some of the above measures of entanglement, as entanglement of formation and distillation, use Bell states as a standard unit of entanglement. This has two disadvantages as pointed out by Nielsen (2000). (a) Measures are not easy to generalize for multipartite entanglement because it is not clear what to take as a standard unit of entanglement in such multipartite systems. (b) Both measures essentially depend, as shown by Nielsen (2000) on the choice of the standard unit of entanglement what, intuitively, should not be the case for good measures of entanglement. They should be "dimensionless".

To deal with this problem Nielsen (2000) suggested three new measures of entanglement: *entanglement of creation*[7], $E_c(\rho)$, *entanglement of communication*[8], $E_{co}(\rho)$, and *entanglement of computation*. Interesting enough, the following identities hold: $E_c = E_f$ and $E_{co} = E_d$.

Nielsen (2000) defines also *entanglement of computation* and conjectures that there may be important connection between this measure of entanglement and key problems of computational complexity. However, his measure of entanglement depends on the choice of some "universal" two qubit gates and just counts the number of these gates needed, in addition to one qubit gates and classical

[7] Entanglement of creation of a state ρ, $E_c(\rho)$ is defined as follows: Alice and Bob wish to create an n copies of ρ having no preshared entanglement, but having possibility to communicate qubits (at a cost) and perform classical communication (at no cost). The entanglement of creation is then defined to be asymptotically minimal value for the ratio of the number of qubits m they transmit to the number n of good (as $n \to \infty$) copies of ρ they create.

[8] Entanglement of communication is defined as follows: Suppose Alice and Bob share n copies of ρ. Then entanglement of communication of ρ, $E_{co}(\rho)$, is defined to be the asymptotically maximal value for the ratio $\frac{m}{n}$ of the number of qubits m Alice can transmit to Bob with asymptotically high fidelity, using LQCC, and n preshared copies of ρ.

communication, to create an entangled state. It is not yet clear which properties such a measure has.

In addition, a variety of new measures of entanglement have been introduced recently in literature. However, it seems to be too early to judge their importance.

Since it is currently not clear enough which approach to quantification of entanglement is the best, the most natural thing to do is to explore available approaches more in depth, and to introduce new ones, till one (some) of them emerges as the proper one.

9 Multipartite Entanglement

Understanding, characterization, classification, quantification as well as manipulation of multipartite entangled states, is a very important and difficult problem, perhaps the main current challenge in quantum information theory, that needs to be explored in order to get more insights into the possibilities of distributed quantum computing and networks. The problem is also of special interest outside of QIP, for physics in general.

Let us start with a simple example illustrating how counterintuitive can be a situation in the case of multipartite entanglement.

In an n-qubit state $|\phi\rangle$ it seems to be natural to say that qubits i and j are entangled if by tracing out the remaining qubits from $|\phi\rangle\langle\phi|$ we get an entangled density matrix. However, in such a case in the GHZ state $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$ no pair of qubits is entangled even though the GHZ state would seem to be maximally entangled. On the other hand in the state $\frac{1}{\sqrt{3}}(|100\rangle + |010\rangle + |001\rangle)$, that seems to be less entangled, actually any two qubits are entangled.

Moreover, as discussed also later, for any integer n and any graph G with the set of vertices $\{1, 2, \ldots, n\}$ there is a quantum state $|\phi_G\rangle$ over H_{2^n} such that qubits i and j are entangled in $|\phi_g\rangle$ if and only if nodes i and j are connected by an edge in G (Dürr, 2000).

9.1 Creation of Multipartite States

There are several techniques how to create multipartite entangled states. A simple idea is due to Bose et al. (1997), as a generalization of the above procedure of entanglement swapping. In the generalized entanglement swapping one deals with generalized n-partite Bell states of the form

$$\frac{1}{\sqrt{2}}(|x\rangle + |\bar{x}\rangle),$$

where $x \in \{0,1\}^n$ and $\bar{x}$ is a bitwise complement of x. Let us assume that we have N sets of entangled particles, each one in a generalized Bell state. Let from the ith set p_i particles are brought together and a joint Bell measurement is performed on them. (Observe that all sets of Bell states of p particles form an orthogonal basis.) In such a case all measured particles get into a Bell state and the same is true for the rest of particles.

Figure 1b shows the case of one GHZ state and two bipartite Bell states. The result of the measurement is one GHZ state and one 4-partite Bell state.

A generalization of the entanglement swapping can also be used to create from an m -partite Bell state and from one GHZ state an $(m+1)$-partite Bell state by simply performing a Bell measurement on one particle of the m-partite state and one particle of the GHZ state, see Figure 1c.

Sometimes a flexibility is needed to create "on demand" a generalized Bell state for a group of particles. In such a case the following method can be used, see Figure 1d. Each party involved share a singlet with a center $\mathcal{O}$. If a set of parties wants to share an multipartite state, then the center performs a generalized Bell measurement of its share of singlets to make the corresponding second particles of parties to get into a generalized Bell state.

In full generality, for qudits and also for the continuous variable case, the problem of entanglement swapping has been solved by Bouda and Bužek (2000).

Experimental creation of multipartite systems have also been attempted: with photon, atoms, ions and nuclear spins at NMR technology. One of them is generation of GHZ states. There has been also experimental creation of a pseudo-pure cat state $\frac{1}{\sqrt{2}}(|0000000\rangle + |1111111\rangle)$.

9.2 Incomparable Forms of Multipartite Entanglement

In the case of bipartite systems one of the fundamental results concerning entanglement is that all entangled states are mutually interconvertible by LQCC (asymptotically, with respect to the number of copies available). Nothing similar is true for multipartite entanglement. It seems that in the case of m-partite systems there are at least $2^m - m - 1$ kinds of asymptotically inequivalent entanglement. (This corresponds, for $n = 2, \ldots, m$, to all $\frac{m!}{n!(n-m)!}$ ways different subsets of n parties can be chosen.) Actually, the understanding of inequivalent ways in which subsystems of a multipartite composite system can entangle is one of the central problems not only of multipartite entanglement, but of all quantum information theory.

The study of entanglement of multipartite systems is at the very beginning even if they are exactly those quantum systems from which one expects the most significant applications — for example for distributed computing, quantum networks or even for understanding of the behaviour of strongly-coupled many-body systems.

Some of the basic problems to deal with are easy to formulate, but usually difficult to solve. For example: How to detect entanglement of multipartite systems? What is the structure of the set of n-partite pure (mixed) entangled states? How many inequivalent types of entanglement we have in multipartite systems? Which sets of entangled states are interconvertible by LQCC (exactly and asymptotically)? Which types of entanglement we have in such multipartite systems (for example what are properties of bound entangled states)? What are the laws entanglement can be shared among n parties? What are proper axioms for measures of entanglement of multipartite systems? Which measures of entanglement of multipartite systems are useful and what are the relations

between them? In general, quantitative and qualitative theory of multipartite entanglement is needed.

So far only very partial results are known, but they well demonstrate complexity of the task.

For example, Dürr et al. (2000) have shown that in an n-partite system, with $n \geq 4$ if two pure states are chosen randomly, then they are *stochastically incomparable* in the sense that neither state can be produced from other by SLQCC. Moreover Bennett et al. (1999) have shown that two GHZ states are LQCC-incomparable with three EPR states.

9.3 Reversible Transformations of Multipartite States and Entropies

In order to study various types of multipartite entanglement the basic question is for which pairs of (sets of) states S and R we can transform S into R in a reversible way using transformations based on LQCC.

For example one GHZ-state can be transformed into one singlet (but not reversibly), and two singlet into one GHZ state (but again not reversibly).

One way to show that some reversible transformations do not exist is to consider how various types of entropies are changed by a transformation process.

It is clear that entropy must be constant during a reversible process, because entropy cannot get increased by such a process.

A useful tool to study reversible transformations has been developed by Linden et al. (1999). They have shown that at any reversible transformations based on LQCC the average relative entropy of entanglement of any pair of parties must be constant. This allowed them to show it is not possible to convert $2n$ GHZ-states into $3n$ singlets, even asymptotically.

An interesting open problem is whether any three-partite state can be transformed in a reversible way into GHZ states and singlets.

9.4 Sharing of Entanglement

Another basic question concerning multipartite entanglement that has already been investigated is the following one: to what extend does entanglement between two objects restrict their entanglement with other objects?

For example, if two qubit-particles are maximally entangled, then they cannot be entangled with other particle (this property of entanglement is usually called *monogamy*). One can therefore expect, in general, that if two particles are entangled, then they can have only limited entanglement with other particles. Let us now discuss several results that have been obtained recently on sharing of entanglement.

Coffmann et al. (1999) considered the following problem: given a pure state of three qubits, say A,B, C, how is tangle between A and B, notation τ_{AB}, related to tangle between A and C, notation τ_{AC}. They showed that

$$\tau_{AB} + \tau_{AC} \leq 4 \det \rho_A.$$

what can be interpreted that tangle between A and B, plus tangle between A and C, cannot be greater than tangle between A and the pair BC.[9]

Bounds on the achievable nearest neighbours entanglement have been obtained also for the case qubits form a bi-infinite translationally invariant linear chain or a finite ring. (Translational equivalence in these cases means that global states to consider are invariant under all shifts of qubits. As a consequence of the translational equivalence, the degree of entanglement between neighbours has to be constant through the chain or ring.) Wooters (2000) has shown that in the case of bi-infinite chains it is possible to achieve an entanglement of formation equal to 0.285 ebits between neighbours and this is the best possible under certain assumptions. Actually, his main result is formulated in terms of concurrence as a measure of entanglement. For the case of rings, O'Connor and Wooters (2000) derived, under certain assumptions, formulas for the maximum possible nearest-neighbour entanglement measured by concurrency.

Sharing of entanglement between n qubits in the case that all pairs of qubits are correlated in the same way, and therefore one can say that entangled pairs of qubits form a complete graph, was investigated by Koashi et al. (2000) and Dürr et al. (2000). They showed that under the above assumptions the maximal entanglement between any pair of qubit, measured in terms of concurrence, is $\frac{2}{n}$.

Dürr (2000) has shown that in the case of n qubits entangled pairs of qubits can form in general any graph of n vertices.

9.5 Measures of Entanglement of Multipartite Systems

One of the key problems concerning multipartite entanglement is that it is far less clear how to measure it than in the bipartite case. How to define appropriate measures of entanglement of multipartite quantum states.

The problem seems to be quite complex. In the case of bipartite systems of two qubits the most natural measures of entanglement are entanglement of formation and distillation. According to these measures the amount of entanglement in a mixed state $|\rho\rangle$ is the number of maximally entangled states (Bell states) that can be distilled from ρ or are needed to form ρ. However, it is not yet clear whether entanglement of multiparticle states can be defined in terms of such a "standard currency". It is not even clear what maximally entangled states are for multipartite systems.

For example, in the case of three qubits it seems that GHZ states and Bell states form a set from which all other pure states can be generated reversibly, but there is no proof of it. (It seems that some other states may be needed.)

One natural approach to explore this problem is to generalize those measures of entanglement for bipartite systems to multipartite systems that are not based on a choice of an unit of entanglement. From this point of view of interests is to

[9] It make sense to speak of the tangle between A and BC, because, even if the state space of BC is four-dimensional, only two of these dimensions are necessary to express pure state of ABC.

generalize relative entropy of entanglement and measures introduced by Nielsen (2000) and discussed above.

An attempt to do such generalization for the entanglement of formation is presented in Wang (2000). Especially entanglement of formation of three-particle systems is studied in depth. Entanglement of a mixed state ρ of a three-partite system $A \otimes B \otimes C$ is defined by

$$E_f(\rho) = \frac{1}{6}[E_f(\rho_{AB}) + S(\rho_{AB}) + E_f(\rho_{AC}) + S(\rho_{AC}) + E_f(\rho_{BC}) + S(\rho_{BC})] + \\ + \frac{1}{6}[S(\rho_A) + S(\rho_B) + S(\rho_C)],$$

where S denotes von Neumann entropy

A (straightforward) generalization of the entanglement of relative entropy has been developed by Plenio and Vedral (2000) and Wang (2000a).

For example, they have shown that for regularized version of the relative entropy of entanglement for three-partite system $A \otimes B \otimes C$, $E_r^\infty(\rho) = \lim_{n\to\infty} \frac{E_r(\rho^{\otimes n})}{n}$ it holds

$$E_r^\infty(\rho) = \frac{1}{3}(E_r^\infty(\rho_{AB}) + E_r^\infty(\rho_{BC}) + E_r^\infty(\rho_{AC})) + \frac{1}{3}(S(\rho_A) + S(\rho_B) + S(\rho_C)).$$

10 Entanglement as a Catalyst

In some cases, for example at teleportation, entanglement can help to perform communication or transmission of quantum states, but in doing that entangled states get destroyed.

Surprisingly enough, there are also cases when "borrowed entanglement" can assist to achieve what without entanglement cannot be done and that "used" entangled states do not get destroyed and at the end of the activities entangled states can be "returned".

10.1 Entanglement Assisted Quantum State Transformations

Jonathan and Plenio (1999) have shown that the mere presence of a (borrowed) entangled state can be an essential advantage when the task is to transform one quantum state into another with local quantum operations and classical communications (LQCC).

They have shown that there are pairs of pure states $(|\phi_1\rangle, |\phi_2\rangle)$ such that using LQCC one cannot transform $|\phi_1\rangle$ into $|\phi_2\rangle$, but with the assistance of an appropriate entangled state $|\psi\rangle$ one can transfer $|\phi_1\rangle$ into $|\phi_2\rangle$ using LQCC in such a way that the state $|\psi\rangle$ is not changed in the process (and can be "returned back" after the process). In such a case $|\psi\rangle$ serves as a "catalyst" for otherwise impossible transformation (reaction).

10.2 Entanglement Assisted Quantum Channel Capacity

A special type of quantum channel capacity has been introduced by Bennett et al. (1999a). Namely, **entanglement assisted classical capacity**, C_E, to be defined as a quantum channel capacity for transmitting classical information with the help of pure entangled states (an unlimited amount of them) shared between the sender and the receiver.

The existence of superdense coding protocol, implies that $C_E(\mathcal{N}) \geq 2C(\mathcal{N})$, for any noiseless quantum channel $\mathcal{N}$, where $C(\mathcal{N})$ is the classical capacity of the channel $\mathcal{N}$.

In the case of noisy quantum channels, surprisingly, $C_E(\mathcal{N})$ can be larger, comparing with $C(\mathcal{N})$, by an arbitrarily large constant factor — even in the case the channel $\mathcal{N}$ is so noisy that $QC_2(\mathcal{N}) = 0$, where $QC_2(\mathcal{N})$ is quantum capacity of the channel $\mathcal{N}$ for the case that two-way classical communication is allowed. By using a noisy quantum channel $\mathcal{N}$ in the superdense coding protocol one can obtain a lower bound on $C_E(\mathcal{N})$. On the other side, by using a noisy classical channel in the teleportation protocol to simulate a quantum channel $\mathcal{N}$ one can obtain an upper bound on $C_E(\mathcal{N})$. This helped to determine exactly $C_E(\mathcal{N})$ for d-dimensional depolarizing and erasure channels. $C_E(\mathcal{N})$ has also been determined exactly for so-called "Bell-diagonal channels" (that commute with superdense coding and teleportation), so that $Tp(Sd(\mathcal{N})) = \mathcal{N}$ (Tp stands for teleportation, ...).

11 Bound Entanglement

One of the most surprising discoveries concerning entanglement of mixed states was due to Horodeckis (1998a). They showed that there is a "free", distillable entanglement; and a "bound" entanglement that is not distillable. That is, there are entangled states from which no pure entangled states can be distilled and therefore such states are not useful for quantum teleportation and (it seems) for quantum communication. Horodeckis (1997a) showed that any entangled mixed state of two qubits can be distilled to get the singlet. However, this is not true in general for entangled states.

(Horodeckis, 1998a) considered two qutrit mixed states

$$\sigma_+ = \frac{1}{3}(|0\rangle|1\rangle\langle 0|\langle 1| + |1\rangle|2\rangle\langle 1|\langle 2| + |2\rangle|0\rangle\langle 2|\langle 0|),$$
$$\sigma_- = \frac{1}{3}(|1\rangle|0\rangle\langle 1|\langle 0| + |2\rangle|1\rangle\langle 2|\langle 1| + |0\rangle|2\rangle\langle 0|\langle 2|)$$

that are separable and using these states they defined a one-parameter family of states

$$\sigma_\alpha = \frac{2}{7}|\Psi^+\rangle\langle\Psi^+| + \frac{\alpha}{7}\sigma_+ + \frac{5-\alpha}{7}\sigma_-,$$

where $|\Psi^+\rangle = \frac{1}{\sqrt{3}}(|0\rangle|0\rangle + |1\rangle|1\rangle + |2\rangle|2\rangle)$, such that these states are separable if $2 \leq \alpha \leq 3$; they are bound entangled if $3 < \alpha \leq 4$ and they are free entangled if $4 \leq \alpha \leq 5$

One of the peculiar features of bound entangled states is that entanglement has to be invested in order to create them by LQCC, but this invested entanglement cannot be distilled/recovered by LQCC.

A first general method to construct states with bound entanglement (bound-entangled states), using the so called unextendible product bases was developed by DiVincenzo et al. (1999).[10] Their family of bound-entangled states depends on 6 parameters. Another construction of a family of bound-entanglement states with 7 parameters, is due to Bruß and Peres (1999).

The goal of attempts to find explicit constructions of bound-entangled states is to help to elucidate properties of such states. For example, to deal with the following problem:

Open problem. Is it true that bound-entangled states satisfy all Bell inequalities and therefore are compatible with the hidden variable interpretation of quantum mechanics?

A simple bound-entangled state in $H_2 \otimes H_2 \otimes H_2 \otimes H_2$ was discovered by Smolin (2000). It is the state (let us call it Smolin's state),

$$\rho_s = \frac{1}{4} \sum_{i=1}^{4} |\Phi_i\rangle\langle\Phi_i| \otimes |\Phi_i\rangle\langle\Phi_i|,$$

where $\Phi_i, i = 1, 2, 3, 4$, are all Bell states. The above state is bound-entangled if the corresponding particles are possesd by four non-cooperating parties (parties in "different labs") — in such a case the parties cannot distill from it a Bell states. However, if we denote parties as A, B, C, D, then the above state is $\{A, B\} : \{C, D\}$, $\{A, C\} : \{B, D\}$ and $\{A, D\} : \{B, C\}$ separable. This is an example of so-called "unlockable-state" that can be used for two of the parties to distill a Bell state provided the other two parties assist them.

An interesting, important and puzzling question remains whether one can make some use of bound-entangled states. In other words, whether there are tasks that can be performed better when assisted by bound-entangled states than with just classically correlated states.

The first natural question along these lines is whether bound-entangled states can be of some use for quantum teleportation. Of course, they are not usable for perfect teleportation because one cannot distill from them maximally entangled pure states. In addition, Linden and Popescu (1998) have shown that with bound-entangled states teleportation cannot be achieved with better than classical fidelity.

Till recently no protocol has been known at which bound-entangled states would performed better than classically entangled states. The first example of some use of bound-entangled states was demonstrated by Murao and Vedral

[10] *An unextendible product basis* (UPB) of H is such an orthonormal set S of product states from H that the subspace H_S generated by S is a proper subspace of H and the orthogonal complement of H_S contains no product state.

(2000). They showed how bound-entanglement can be used for **remote information concentration** — the reverse process to quantum telecloning.

In the information concentration scheme there are four parties, Alice, Bob, Charles and David and they share four particles in Smolin's state. The particles of Alice, Bob and Charles represent their input port qubits and that of David the output port qubit. Bob and Charles posses perfect clones of an unknown qubit state $|\phi\rangle$. At the end of the remote information concentration scheme David's particle gets into the state $|\phi\rangle$. In the protocol Alice, Bob and Charles first perform Bell measurement between their ancilla or optimal clones and their particles of the Smolin state. They communicate results of their measurement to David (who gets in total 6 bits), who then performs a product (determined uniquely by six communicated bits) of Pauli matrices on his particle to get the state $|\phi\rangle$.

However, bound entangled states can be *activated* (Horodeckis, 1998b) and *superactivated* (Shor et al., 2000).

Activation of bound entanglement, or the so called *quasi-distillation*, refers to the process in which a finite number of free entangled mixed states are distilled with the assistance of a number of bound entangled states, but without such an assistance no useful entanglement can be distilled for these states. In a superactivation two bound entangled states are combined (tensored) to get a state which is not bound entangled – in other words, in this case bound entangled states are activated by bound entangled states — and therefore it can be distilled. Several examples of superactivation discuss Dürr and Cirac (2000).

Bound entangled states have various interesting properties. Let us summarize some of them.

- A bound entangled state remains bound entangled when subjected to any LQCC.
- There is no bound entangled state of two qubits.
- Each PPT-state is bound entangled and it is an open problem whether there are also bound entangled states that are NPT-states.[11]

In spite of a variety results concerning bound entanglement the very fundamental question remains open: What is the role of bound entanglement in Nature? It seems useless for communication, but this does not have to be the whole story. Another fundamental question is about nonlocality of bound entangled states?

12 Quantum Information Processing Principles

The thesis that information is physical and that the role of information in physics is analogous to that of energy in thermodynamics leads naturally to the search

[11] DiVincenzo et al. (1999) exhibit a canonical two-parameter family $\mathcal{S}$ of bipartite mixed states in Hilbert space $H_d \otimes H_d$, which are NPT-states and there is an evidence that from them one cannot distill using LQCC maximally entangled states of two qubits. These states are canonical in the sense that any bipartite NPT state can be reduced by LQCC to a state in $\mathcal{S}$.

for information processing principles and laws. For example, for principles and laws analogous to those in thermodynamics. It is only natural that quantum entanglement is expected to play the key role in such principles and laws. One such emerging principles seems to be **no-increasing of entanglement principle** — under local quantum operations and classical communications — (Horodeckis, 1997).

Informally, this principle says that entanglement cannot increase under local operations and classical communication. More formally, it says that if E is a "proper" measure of entanglement and an input state ρ is transformed by quantum operations into a mixed state $\bigoplus_{i=1}^{k}\{(p_i, \rho_i)\}$, then $\sum_{i=1}^{k} p_i E(\rho_i) \leq E(\rho)$.

This principle has been proven to hold for entanglement of formation (see Bennett et al. 1997a). A form of this principle has been proven to be equivalent with no-cloning theorem.

12.1 Thermodynamics and Entanglement

The above entanglement processing principles are connected with a belief of some that there are laws of governing entanglement processing that are analogous to those of thermodynamics. The entanglement is considered then to play the role of energy. This entanglement-energy analogy has been farther explored recently by Horodeckis (2000). They postulate that

- Entanglement is a form of quantum information corresponding to internal energy;
- Sending qubits corresponds to work.

They have also shown that in the closed bipartite quantum communication systems, information is conserved and so is in deterministic distillation protocols.

In order to consider balance of quantum information they introduce, in addition to the concept of physical work, the concept of **useful logical work** in quantum communication, to be the amount of qubits transmitted without decoherence.

13 Entanglement Based Quantum Cryptography

In the rest of the paper we discuss some of the areas where entanglement seems to have potential to yield significant practical applications. The first such area is quantum cryptography.

Quantum key generation (QKG) is another area where entanglement has been shown to be of large importance.

There is in principle a very easy and clearly unconditionally secure way for two parties, Alice and Bob, to make use of entanglement to generate a random key.

Let Alice and Bob share n pairs of particles, each pair in the entangled state

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

n pairs of particles in EPR state

If both parties measure their particles in the standard basis (and it does not matter in which order), they receive, as the result of their measurement, the same random binary string of length n. This way of binary key generation is absolutely secure because no information is transmitted.

The catch is that if Alice wants to generate with a party a random key, it is non-trivial to achieve that they share n pairs of particles in the above pure entangled state. This can be done, using one of the purification techniques discussed above, but it is a complex process and both parties need to posses a quantum processor in order to do that.

The above way of generating random key is also a good way to explain why we say that entanglement does not allow a communication. In the case of two parties, Alice and Bob, we say that Alice has communicated with Bob if she did something what had an effect on measurements at Bob side. In the case of the above protocol Alice and Bob knows at the end that they have the same key, but none of them has a way to influence which one.

13.1 Ekert's Protocol and Its Generalizations

In 1991 Ekert discovered a new type of QKG protocols, security of which is based on the completeness of quantum mechanics. Completeness here means that quantum mechanics provides maximum possible information about any quantum system and an eavesdropping can then be seen as "an introducing some elements of physical reality". To each specific entanglement-based QKG protocol a (generalized) Bell inequality is formed and by a statistical test of this inequality, on the basis of the probability statistics on rejected data, it is possible to determine whether eavesdropping took place. (Eve plays here the role of a hidden variable and disturbs the quantum nature of the correlations between the probabilities of occurrence of various outcomes for various measurements.)

Example. Let a source emit spin-$\frac{1}{2}$ particles in the state $\frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$. Alice performs her measurement with respect to the angles $0°$, $45°$, $90°$ and Bob with respect to angles $45°$, $90°$ and $135°$.

Denote $E(i, j, b_1, b_2)$ the probability that if Alice measures with respect to vector α_i, Bob with respect to β_j, then Alice (Bob) gets as the outcome b_1 (b_2).

Let us also denote

$$E(i,j) = E(i,j,1,1) + E(i,j,0,0) - E(i,j,1,0) - E(i,j,0,1).$$

In such a case quantum mechanics requires that for

$$E(1,1) - E(1,3) + E(3,1) + E(3,3) = S$$

it holds $S = -2\sqrt{2}$ but if an eavesdropping introduces an element of physical reality, then it holds

$$-2 \leq S \leq 2.$$

13.2 Entanglement and Unconditional Security of Cryptographic Protocols

The positive role entanglement can play for quantum key generation, to the extend that we can have unconditionally secure quantum key generation, created a hope that we can have unconditional security also for such basic cryptographic primitives as (exact) quantum coin tossing and especially for bit commitment. However, this has turned out not to be the case. It has ben proven that unconditionally secure quantum bit commitment is not possible. Cheating is always possible – due to the entanglement. This time it is just entanglement playing the role of the bad guy.

13.3 Quantum Secret Sharing, Strong Quantum Secret Sharing and Quantum Cobweb

Several variants of the classical secret sharing problem have been considered

Secret Sharing. There is a simple method, due to Hillery, Bužek and Berthiaume (1998), how Alice can distribute a (secret) qubit $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ to Bob and Charles in such a way that they have to cooperate in order to get $|\phi\rangle$.

The basic idea is that Alice couples a given particle P in the state $|\phi\rangle$ with the GHZ state $|\psi\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$ of three particles P_a, P_b and P_c she shares with Bob and Charles and then performs a measurement on the state of particles P and P_a, with respect to the Bell basis $\{\Phi^{\pm}, \Psi^{\pm}\}$. Since

$$\begin{aligned}|\phi\rangle|\psi\rangle = \frac{1}{2}(&|\Phi^+\rangle(\alpha|00\rangle + \beta|11\rangle) + |\Phi^-\rangle(\alpha|00\rangle - \beta|11\rangle)\\ &+|\Psi^+\rangle(\beta|00\rangle + \alpha|11\rangle) + |\Psi^-\rangle(-\beta|00\rangle + \alpha|11\rangle)),\end{aligned}$$

the outcome of the measurement is that particles P_b and P_c get into one of the states

$$\frac{1}{\sqrt{2}}(\alpha|00\rangle + \beta|11\rangle), \frac{1}{\sqrt{2}}(\alpha|00\rangle - \beta|11\rangle), \frac{1}{\sqrt{2}}(\beta|00\rangle + \alpha|11\rangle), \frac{1}{\sqrt{2}}(-\beta|00\rangle + \alpha|11\rangle)$$

and Alice gets two bits to tell her about which of these four cases happened. However, neither Bob nor Charles has information about which of these four states their particles are in.

Bob now performs a measurement of his particle with respect to the dual basis. He gets out of it one bit of information and Charles's particle P_c gets into one of 8 possible states, which is uniquely determined by bits both Alice and Bob got as the results of their measurements, and which can be transformed into the state $|\phi\rangle$ using one or two applications of Pauli matrices. Secret sharing was experimentally verified by Tittel et al. (2000).

A more general problem is to design methods for the so called quantum (t, n) threshold sharing schemes. That is how one can "partition" or "share" a quantum "secret" (state) among n parties in such a way that, for a given t, any t of parties can, by cooperation, reconstruct the secret, but no combination of $t-1$ of parties is able to do that.

This problem has been solved in full generality by Cleve et al. (1999). They showed that such secret sharing protocol exists if $k > \frac{n}{2}$. This limitation is due to No-cloning theorem.

Classicaly, also more general secret sharing schemes are considered at which the so called *access structure* - sets of users that should be able to get secret – is specified. Monotonicity of the access structures is the only requirement for the existence of classical secret sharing schemes. Gottesman (1999) has extended the results of Cleve et al. (1999), to consider also more general access structures with the outcome that monotonicity and No-cloning theorem provide the only restrictions on the access structures.

Secret Sharing in a Strong Sense. In the secret sharing scheme mentioned above there is no guarantee that shared secret remains really secret if two involved parties start to communicate classically. Moreover, as shown by Walgate et al (2000), there is no way to hide a bit between two parties if classical communication is possible and shares consist of parts of two orthogonal pure states.

Let us say that a *secret is shared by two parties in a strong sense* if classical communication between parties is not sufficient for them to get the secret.

Terhal et al. (2000) have shown that it is possible to share a single bit, using mixed quantum states, between two parties in such a way that using classical communication they can obtain only arbitrarily little information about the bit, but they can get the bit only by joint quantum measurement. Moreover, surprisingly, not too much entanglement is needed to share/hide one bit.

The protocol suggested by Terhal et al. (2000) is surprisingly simple and the integer n used in the protocol is the degree of security of the protocol. The hider Alice chooses uniformly and randomly n Bell states with the only restriction that the number of singlets between chosen states is even (odd) if the bit to hide is 0 (1). Alice then distributes the chosen states to Bob and Charles, in such a way that the first (second) particle of each entangled pair goes to Bob (Charles).

Surprisingly, it can be shown that the state shared by Alice and Bob has actually much less entanglement as the above protocol seems to indicate. Indeed, the state hiding the bit 0 can be prepared without the use of the entanglement and the state hiding the bit 1 can be prepared using just one singlet.

Quantum Cobweb. Pati (2001) has discovered another surprising use of entanglement. He has shown a way how a secret (an unknown qubit) can be diluted between n remotely located parties being in a special entangled state (amplitudes of which sum to zero), in such a way that these parties cannot disclose secret using (joint) unitary operations, no matter how they cooperate. This type of quantum channel where an unknown qubit once enters remains interwinded with multipartites is sort of "a quantum cobweb". (However, it is not clear whether such a state could be recovered by EALQCC.)

Acknowledgment

This is to thank to Petr Macháček for careful reading, improving and commenting the manuscript.

References

1. L. Accardi and M. Ohya. Teleportation of general quantum systems. Technical report, quant-ph/9912087, 1999.
2. S. Albeverio and S.-M. Fei. Teleportation of general finite dimensional quantum systems. Technical report, quant-ph/0012035, 2000.
3. J. S. Bell On the Einstein-Podolsky-Rosen paradox. *Physics 1*, pages 195–200, 1964. Reprinted in Quantum Theory and Measurement, (eds): J. A. Wheeler and W. H. Zurek, 403-408.
4. C. H. Bennett, D. DiVincenzo, J. Smolin, B. Terhal, and W. Wooters. Remote state preparation. Technical report, quant-ph/0006044, 2000.
5. C. H. Bennett, S. Popescu, D. Rohrlich, J. A. Smolin, and A. V. Thapliyal. Exact and asymptotic measures of multipartite pure state entanglement. Technical report, quant-ph/9908073, 1999.
6. C. H. Bennett, P. W. Shor, J. A. Smolin, and A. V. Thapliyal. Entanglement-assisted classical capacity of noisy quantum channels. Technical report, quant-ph/9904023, 1999a.
7. C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters*, 70:1895–1899, 1993.
8. C. H. Bennett, G. Brassard, S. Popescu, B. W. Schumacher, J. A. Smolin, and W. K. Wootters Purification of noisy entanglement and faithful teleportation via noisy channels. *Physical Review Letters*, 76(5):722–725, 1996.
9. C. H. Bennett, D. P. DiVincenzo, C. A. Fuchs, T. Moric, E. M. Rains, P. W. Shor, and J. A. Smolin Quantifying non-locality without entanglement. Technical report, quant-ph/9804053, 1998.
10. C. H. Bennett, D. P. DiVincenzo, and J. A. Smolin Capacities of quantum erasure channels. Technical report, quant-ph/9701015, 1997a.
11. C. H. Bennett, D. P. DiVincenzo, J. A. Smolin, and W. K. Wootters Mixed state entanglement and quantum error correction. *Physical Review A*, 54:3824–3851, 1996a. quant-ph/9604024.
12. D. Bohm. A suggested interpretation of the quantum theory in terms of "hidden" variables I and II. *Physics Review*, 85:166–193, 1952. also in "Quantum theory and measurements" (ed. J. A. Wheeler and W. H. Zurek), Princeton University Press, 1983.

13. S. Bose and D. Home. A generic source of entanglement showing complementarity between entanglemen and particle distinguishibility. Technical report, quant-ph/0101093, 2000.
14. S. Bose and V. Vedral. Mixedeness and teleporattion. Technical report, quant-ph/9912033, 1999.
15. S. Bose, V. Vedral, and P. L. Knight. A multiparticle generalization of entanglement swapping. Technical report, quant-ph/9708004, 1997.
16. J. Bouda and V. B. zek. Entanglement swapping between multi-qudit systems. Technical report, Tech. Report, Faculty of Informatics, Masaryk University, 2000.
17. D. Bouwmeester, J.-W. Pan, K. Mattle, M. Eibl, H. Weinfurter, and A. Zeilinger. Experimental quantum teleportation. *Nature*, 390(6660):575–579, December 1997.
18. G. Bowen. Classical information capacity of superdense coding. Technical report, quant-ph/0101117, 2001.
19. G. Brassard. Quantuqm communication complexity. Technical report, quant-ph/0101005, 2001.
20. D. Bruß and A. Peres. Construction of quantum states with bound entanglement. Technical report, quant-ph/9911056, 1999.
21. H. Buhrman, R. Cleve, and A. Wigderson. Quantum vs. classical communication and computation. In *Proceedings of* 30*th ACM STOC*, pages 63–68, 1998. quant-ph/9802040.
22. R. Cleve, D. Gottesman, and H.-K. Lo. How to share a quantum secret. Technical report, quant-ph/9901025, 1999.
23. V. Coffman, J. Kundu, and W. K. Wootters. Distributed entanglement. Technical report, quant-ph/9907047, 1999.
24. D. P. DiVincenzo, T. Mor, P. W. Shor, J. A. Smolin, and B. M. Terhal. Unextendible product bases, uncompletable product bases, and bound entanglement. Technical report, quant-ph/9908070, 1999.
25. D. P. DiVincenzo, P. W. Shor, J. A. Smolin, B. M. Terhal, and A. V. Thapliyal. Evidence for bound entangled states with negative partial transpose. Technical report, quant-ph/9910026, 1999.
26. D. P. DiVincenzo, C. A. Fuchs, H. Mabuchi, J. A. Smolin, A. Thapliyal, and A. Uhlmann. Entanglement of assistance. Technical report, quant-ph/9803033, 1998.
27. W. Dürr. Entanglement molecules. Technical report, quant-ph/0006105, 2000.
28. W. Dürr and J. Cirac. Multipartite entanglement and its experimental detection. Technical report, quant-ph/0011025, 2000.
29. A. Einstein, B. Podolsky, and N. Rosen. Can quantum mechanical description of physics reality be considered complete? *Physical Review*, 47:777–780, 1935.
30. A. K. Ekert Quantum cryptography based on Bell's theorem. *Physical Review Letters*, 67(6):661–663, 1991.
31. B. Enquist and W. Schmid, editors. *Mathematics unlimited, 2001 and beyond*, chapter Quantum computing challenges, pages 529–564. Springer, 2000.
32. V. N. Gorbachev and A. I. Trubilko. Quantum teleportation of EPR pair by three-particle entanglement. Technical report, quant-ph/9906110, 1999.
33. V. N. Gorbachev, A. I. Zhiliba, A. I. Trubilko, and E. S. Yakovleva. Teleportation of entangled states and dense coding using a multiparticle channel. Technical report, quant-ph/0011124, 2000.
34. D. Gottesman. Theory of quantum secret sharing. Technical report, quant-ph/9910067, 1999.

35. D. Gottesman and I. L. Chuang. Quantum teleportation is a universal computational primitive. Technical report, quant-ph/9908010, 1999.
36. D. M. Greenberger, M. A. Horne, and A. Zeilinger. *Bell's theorem and the conception of the universe*, chapter Going beyond Bell's theorem, pages 69–. Kluwer Academic, Dordrecht, 1989.
37. J. Gruska. *Quantum computing*. McGraw-Hill, 1999. See also additions and updatings of the book on http://www.mcgraw-hill.co.uk/gruska.
38. L. Hardy. Spooky action at a distance in quantum mechanics. *Contemporary physics*, 6(6):419–429, 1998.
39. L. Hardy. Disentangling non-locality and teleportation. Technical report, quant-ph/9906123, 1999.
40. M. Hillery, V. Bužek, and A. Berthiaume. Quantum secret sharing. Technical report, quant-ph/9806063, 1998.
41. S. Hill and W. K. Wootters Entanglement of a pair of quantum bits. Technical report, quant-ph/9703041, 1997.
42. T. Hiroshima. Optimal dense coding with mixed state entanglement. Technical report, quant-ph/0009048, 2000.
43. M. Horodecki, P. Horodecki, and R. Horodecki. Inseperable two spin 1/2 density matrices can be distilled to a singlet form. *Physics Review Letters*, 78:547–577, 1997.
44. M. Horodecki, P. Horodecki, and R. Horodecki. Separability of mixed states: necessary and sufficient conditions. *Physics Letters A*, 223:1–8, 1998. quant-ph/9605038.
45. M. Horodecki, P. Horodecki, and R. Horodecki. Limits for entanglement measures. Technical report, quant-ph/9908065, 1999.
46. M. Horodecki, P. Horordecki, and R. Horodecki. *Quantum information - an introduction to basic theoretical concepts and experiments*, chapter Mixed-state entanglement and quantum communication. Springer, 2001.
47. R. Horodecki, M. Horodecki, and P. Horodecki. Reversibility of local transformations of multipartite entanglement. Technical report, quant-ph/9912039, 1999.
48. R. Horodecki, M. Horodecki, and P. Horodecki. On balance of information in bipartite quantum information systems: entanglement-energy analogy. Technical report, quant-ph/0002021, 2000.
49. M. Horodecki, P. Horodecki, and R. Horodecki. Mixed-state entanglement and distillation: is there a "bound" entanglement in nature? Technical report, quant-ph/9801069, 1998a.
50. M. Horodecki, P. Horodecki, and R. Horodecki. Bound entanglement can be activated. Technical report, quant-ph/9806058, 1998b.
51. M. Horodecki and R. Horodecki. Are there basic laws of quantum information processing? Technical report, quant-ph/9705003, 1997.
52. J. G. Jensen and R. Schack. Quantum authentication and key distribution using catalysis. Technical report, quant-ph/00031104, 2000.
53. I. D. Jonathan and M. B. Plenio. Entanglement-assisted local manipulation of pure quantum states. Technical report, quant-ph/9905071, 1999. See also Phys. Rev. Lett., 83, 3566-3569, 1999.
54. R. Jozsa, D. S. A. J. P. Dowling, and C. P. Williams. Quantum clock synchronization based on shared prior entanglement. Technical report, quant-ph/0004105, 2000.
55. P. Kaye and M. Mosca. Quantum networks for concentrating entanglement. Technical report, quant-ph/0101009, 2001.

56. A. Kent. Entangled mixed states and local purification. Technical report, quant-ph/9805088, 1998.
57. A. Kent, N. Linden, and S. Massar. Optimal entanglement enhancement for mixed states. Technical report, quant-ph/9902022, 1999.
58. Y.-H. Kim, S. P. Kulik, and Y. Shih. Quantum teleportation with a complete Bell state measurement. Technical report, quant-ph/0010046, 2000.
59. M. Koashi, V. B. zek, and N. Imoto. Tight bounds for symmetric sharing of entanglement among qubits. Technical report, quant-ph/0007086, 2000.
60. B. Kraus and J. I. Cirac. Optimal creation of entanglement using a two-qubit gate. Technical report, quant-ph/0011050, 2000.
61. M. Lewenstein, B. Kraus, J. I. Cirac, and P. Horodecki. Optimization of entanglement witnesses. Technical report, quant-ph/0005014, 2000.
62. N. Linden and S. Popescu. Bound entanglement and teleportation. Technical report, quant-ph/9807069, 1998.
63. N. Linden, S. Popescu, B. Schumacher, and M. Westmoreland. Reversibility of local transformations of multiparticle entanglement. Technical report, quant-ph/9912039, 1999.
64. L. Marinatto and T. Weber. Which kind of two-particles states can be teleported through a three-particle quantum channel? Technical report, quant-ph/0004054, 2000.
65. M. Murao and V. Vedral. Remote information concentration using a bound entangled state. Technical report, quant-ph/0008078, 2000.
66. M. A. Nielsen. On the units of bipartite entanglement: is sixten ounces of entanglement always equal to one pound? Technical report, quant-ph/0011063, 2000.
67. M. A. Nielsen and J. Kempe. Separable states are more disordered globally than locally. Technical report, quant-ph/0011117, 2000.
68. M. Nielsen A partial order on the entangled states. Technical report, quant-ph/9811053, 1998.
69. K. M. O.Connors and W. K. Wooters. Entangled rings. Technical report, quant-ph/0009041, 2000.
70. A.-W. Pan, D. Bouwmeester, M. Daniell, H. Weinfurter, and A. Zeilinger. Experimental test of quantum non-loclaity in three photon Greenberg-Horne-Zeilinger entanglement. *Nature*, 403:515–519, 2000.
71. J. W. Pan, D. Bouwemester, H. Weinfurter, and A. Zeilinger. Experimental entanglement swapping: entangled photons that never interacted. *Physical Review Letters*, 80:3891–3894, 1998.
72. J.-W. Pan, C. Simon, Č. Brukner, and A. Zeilinger. Feasible entanglement purification for quantum communication. Technical report, Institut für Experimentalphysik, universität Wien, 2000.
73. A. K. Pati. Quantum cobweb: remote shared-entangling of an unknown quantum state. Technical report, quant-ph/0101049, 2001.
74. I. C. Percival. Why Bell experiments. Technical report, quant-ph/0008097, 2000.
75. A. Peres. Separability criterion for density matrices. *Physical Review Letters*, 77:1413–1415, 1996a.
76. A. Peres. Delayed choice for entanglement swapping. *Journal of Modern Optics*, 47:139–143, 2000.
77. M. Plenio and V. Vedral. Bounds on relative entropy of entanglement for multiparty systems. Technical report, quant-ph/0010080, 2000.
78. M. B. Plenio and V. Vedral. Teleporatation, entanglement and thermodynamics in the quantum world. Technical report, quant-ph/9804075, 1998.

79. R. Rausschendorf and H. Briegel. Quantum computing via measurements only. Technical report, quant-ph/0010033, 2000.
80. R. Raz. Exponential separation of quantum and classical communication complexity. In *Proceedings of* 31*st ACM STOC*, pages 358–367, 1999.
81. O. Rudolp. A new class of entanglement measures. Technical report, quant-ph/0005011, 2000.
82. E. Schrödinger. Die gegenwartige situation in der quanenmechanik. *Natürwissenschaften*, 23:807–812, 823–828, 844–849, 1935.
83. P. Shor and J. S. an A. Thapliyal. Superactivation of bound entanglement. Technical report, quant-ph/0005117, 2000.
84. J. A. Smolin. A four-party unlockable bound-entangled state. Technical report, quant-ph/0001001, 2000.
85. H. P. Stapp. From quantum non-locality to mind-brain interactions. Technical report, quant-ph/0010029, 2000.
86. A. M. Steane and W. van Dam. Guess my number. *Physics Today*, 53(2):35–39, 2000.
87. B. M. Terhal. Detecting quantum entanglement. Technical report, quant-ph/0101032, 2001.
88. B. M. Terhal, D. P. DiVincenzo, and D. W. Leung. Hidding bits in Bell states. Technical report, quant-ph/0011042, 2000.
89. W. Tittel, T. Brendel, H. Zbinden, and N. Gisin. Quantum cryptography using entangled photons in energy-time Bell states. *Physics Review Letters*, 84:4737–4740, 2000.
90. W. Tittel, H. Zbinden, and N. Gisin. Quantum secret sharing using pseudo-GHZ states. Technical report, quant-ph/9912035, 1999.
91. W. Tittel, J. Brendel, H. Zbinden, and N. Gisin. Experimental demonstration of quantum correlation over more than 10km. *Physical Review A*, 57(5):3229–3232, 1998. Preprint quant-ph/9806043.
92. A. Uhlmann. Entropy and optimal decomposition of states relative to a maximal commutative algebra. Technical report, quant-ph/9704017, 1997.
93. L. Vaidman. Teleportation: dream or reality? Technical report, quant-ph/9810089, 1998.
94. W. van Dam, P. Høyer, and A. Tapp. Multiparty quantum communication complexity. Technical report, quant-ph/9710054, 1997.
95. V. Vedral and M. B. Plenio Entanglement measures and purification procedures. Technical report, quant-ph/9707035, 1997.
96. J. Walgate, A. J. Short, L. Hardy, and V. Vedral. Local distinguishability of multipartite orthogonal quantum states. Technical report, quant-ph/0007098, 2000.
97. A. M. Wang. Bounds on the generalized entanglement of formation for multiparity systems. Technical report, quant-ph/0011091, 2000.
98. A. M. Wang. Improved relative entropy of entanglement for multi-party systems. Technical report, quant-ph/0012029, 2000a.
99. R. F. Werner. All teleportation and dense coding schemes. Technical report, quant-ph/0003070, 2000.
100. W. K. Wootters. Entangled chains. Technical report, quant-ph/0001114, 2000.
101. M. Žukowski, A. Zeilinger, M. A. Horne, and A. K. Ekert. "Event-ready-detectors"; Bell experiments via entanglement swapping. *Physical Review Letters*, 71:4287–4290, 1993.

Combinatorial and Computational Problems on Finite Sets of Words

Juhani Karhumäki *

Department of Mathematics and Turku Centre for Computer Science,
University of Turku, FIN-20014 Turku, Finland
karhumak@cs.utu.fi

Abstract. The goal of this paper is to give a survey on recent work studying several combinatorial and computational problems on finite sets of words. The problems are natural extensions of three well-understood problems on words, namely those of the *commutation*, the *conjugacy*, and the *equivalence of morphisms on regular languages*. Each of these problems provides challenging questions in the case of finite sets of words.

1 Introduction

Combinatorics on words is a relatively new area on discrete mathematics related and motivated by theoretical computer science. It is an enormous source of fascinating problems, see e.g. [Ka]. Some of the problems are very difficult, and their solutions have become jewels of discrete mathematics. The satisfiability problem of word equations is such an example, see [Ma], [Pl], and [Di] for its solution.

Some other problems are not difficult to prove, but have turn out extremely fruitful for many applications, for example in efficient string matching algorithms, cf. [CR]. Examples of such basic results of words are, for instance, the characterization when two words commute and the periodicity lemma of Fine and Wilf, cf. [ChK].

In this paper we consider three well-understood problems on words, and try to define and analyze what can be said about their natural extensions for finite sets of words. The problems are *the commutation problem*, the *conjugacy problem*, and the *equivalence problem of morphisms on regular languages*. Consequently, in the first and the second problem we ask to characterize when two elements commute or are conjugates, respectively, while in the third problem we ask for an algorithm to decide whether two morphisms h and g are equivalent on all words of a given regular language L, i.e., $h(w) = g(w)$ holds for all w in L. Each of these problems have a simple solution.

The above three problems can be formulated, in a natural way, for finite sets of words. However, the nature of the problems changes drastically: none of these problems remain simple, and only partial solutions are known. Even more interestingly several challenging variants of these problems can be stated.

* Supported under the grant #44087 of the Academy of Finland

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 69–81, 2001.

This is the point we want to make in this paper. In very general terms we are looking at what happens to certain simple problems when a deterministic set-up is extended to a nondeterministic one, when words are replaced by finite sets of words.

Most of the results of this paper have been achieved over the past few years in a collaboration with several researchers, see [CKO], [CKM], [KP], [KLII], and [HIKS].

2 Preliminaries

In this section we fix the terminology and notations used later, for more on words we refer to [Lo] or [ChK] and on automata and formal languages to [HU].

A finite *alphabet* is denoted by A, and the *free semigroup* (resp. *free monoid*) it generates by A^+ (resp. A^*). Elements (resp. subsets) of A^* are called *words* (resp. *languages*). The empty word is denoted by 1. We are mainly dealing with finite languages. The *monoid of finite languages* under the operation of the product is denoted by $\mathcal{P}(A^*)$. We use lower case letter u, v, w, ... to denote words and capital letter U, V, W, ... to denote languages. For unknowns assuming words (resp. languages) as their values we use letters x, y, z, ... (resp. X, Y, Z, ...).

A *morphism* from A^* into B^* is a mapping h satisfying $h(uv) = h(u)h(v)$ for all words u and v in A^*. Its nondeterministic variant, a *finite substitution*, is a morphism $\varphi : A^* \to \mathcal{P}(B^*)$. Let $L \subseteq A^*$ be a language and $h, g : A^* \to B^*$ two morphisms. We say that *h and g are equivalent on L* if

$$h(w) = g(w) \text{ for all } w \text{ in } L. \tag{1}$$

Similarly, we can define the equivalence of finite substitutions $\varphi, \psi : A^* \to \mathcal{P}(B^*)$ on a language L, when the equality in (1) is, of course, the set equality. More generally, a monoid B^* above can be replaced not only by the monoid $\mathcal{P}(B^*)$, but also by an arbitrary monoid.

Some of our problems can be stated in terms of equations over a monoid. Let Ξ be a finite set of *unknowns* and M a monoid. An *equation with unknowns in Ξ* is a pair $(\alpha, \beta) \in \Xi^+ \times \Xi^+$ and its *solution in M* is a morphism $s : \Xi^* \to M^*$ satisfying $s(\alpha) = s(\beta)$ in M. Accordingly, we define systems of equations and their solutions. We say that two systems of equations are *equivalent* in M if they have exactly the same solutions in M, and that a system is *independent* in M if it is not equivalent in M to any of its proper subsystems.

3 Problems

In this section we formulate our three problems. We need to recall that two elements α and β of a monoid M *commute* if

$$\alpha\beta = \beta\alpha$$

and *are conjugates* if

$$\exists \gamma \in M : \quad \alpha\gamma = \gamma\beta.$$

The following problems are very natural:

Problem 1 (The Commutation Problem). Characterize when two elements of a monoid M commute.

Problem 2 (The Conjugacy Problem). Characterize when two elements of a monoid M are conjugates.

Our third problem is a decision question:

Problem 3 (The equivalence of morphisms on regular languages). Given a regular language $L \subseteq A^*$ and morphisms $h, g : A^* \to M$. Decide whether or not h and g are equivalent on L.

The choice of L here might look rather arbitrary. However, from the point of view of this paper there exist two good reasons to assume that L is regular. First, it makes Problem 3 easily decidable for free monoids B^*. Second, and more importantly, it makes the problem very interesting for the monoid of finite languages, that is for finite substitutions.

Of course, by choosing the language L in a more complicated way, Problem 3, even in the case of free monoids, can be made difficult, or even open. This, however, is outside of the scope of this paper.

4 The Word Case

In this section we recall the solutions of Problems 1-3 for word monoids A^*.

Proposition 1. *The following conditions are equivalent for words u and v in A^*:*

(i) there exists a word p such that $u, v \in p^*$;
(ii) u and v commute, i.e., the pair (u, v) satisfies the equation $xy = yx$;
(iii) the pair (u, v) satisfies a nontrivial equation.

A proof of this important result, by induction based on the length of uv, requires only a few lines. The equivalence of (i) and (ii) gives a simple combinatorial characterization for the commutation of two words. The equivalence of (i) and (iii), in turn, states a special case of the well known defect theorem, see e.g. [ChK]: if two words satisfy a nontrivial relation, then these words can be expressed as a product of a single word, i.e., as a power of a single word. It is interesting to note that natural attempts to extend this result leads to an open problem on words: it is not known whether or not any independent system of equations with three unknowns and three equations has only periodic solutions, i.e., all components are powers of some word. For two equations this is not the case, cf. e.g. [ChK].

Similarly, the conjugacy of words has a combinatorial characterization:

Proposition 2. *The following conditions are equivalent for words u and v in A^+:*

(i) there exist words p and q such that $u = pq$, $v = qp$:
(ii) there exists a word w such that $uw = wv$, i.e., the triple (u, w, v) is a solution of the equation $xz = zy$;
(iii) there exist p and q such that for some w, $u = pq$, $v = qp$ and $w \in (pq)^*p$.

By (i) the conjugacy of words means that one is obtainable from the other by a cyclic permutation, i.e., by moving a prefix of the first word to its end. Condition (ii), in turn, shows that the property "being conjugates" is expressible as a set of all solutions of a word equation. Here an extra unknown, namely z, is needed while in the similar characterization for the commutation no such unknown is necessary. A systematic research on expressive power of word equations was initiated in [KMP]. Finally, the condition (iii) gives the full solution of the conjugacy equation $xz = zy$.

A solution to Problem 3 is based on the following result.

Proposition 3. *Let $L \subseteq A^*$ be a regular language accepted by an n-state finite automaton. Then for any morphisms $h, g : A^* \to B^*$ the following conditions are equivalent:*

(i) $h(w) = g(w)$ for all $w \in L$;
(ii) $h(w) = g(w)$ for all $w \in F = \{w \in L \mid |w| \leq 2n\}$.

The equivalence of (i) and (ii) is based on the pumping lemma of regular languages and a simple combinatorial property of words. Proposition 3 immediately implies an affirmative answer to Problem 3 for morphisms of free monoids. Actually, it states even more, namely that any regular language L possesses a *finite test set* F which, moreover, can be found effectively, i.e., a finite subset F of L such that to test whether two morphisms are equivalent on L it is enough to test that on F. An important and fundamental property (the *Ehrenfeucht Compactness Property*) of free monoids asserts that any language possesses a finite test set, but, in general, it needs not be effectively findable, see [Gu], [AL], and [ChK].

5 The Finite Language Case

In this main section we consider the above problems for finite sets of words. As we shall see the problems become much more intriguing, in particular in many cases even very restricted variants of these problems are of great interest. We consider each of the problems separately.

5.1 The Commutation

We consider the equation

$$XY = YX \tag{2}$$

on the monoid of languages. Our main interest is in the case when languages, or at least one of those, is finite, although one of our main problems is formulated for regular languages. We concentrate on the case when languages are subsets of A^+, i.e., not containing the empty word. It follows immediately that for a fixed X there exists the unique maximal set commuting with X, and moreover it is a semigroup. It is obtained as the union of all sets commuting with X and referred to as the *centralizer* of X, in symbols $\mathcal{C}(X)$. It was already in 1970 when Conway posed the following problem:

Problem 4 (Conway's Problem). Is the centralizer of a regular set also regular?

Surprisingly, this has turn out to be very intriguing problem. Indeed, even the much simple problem seems to be still unanswered:

Problem 5. Is the centralizer of a regular (or even finite) set recursive?

It is interesting to note that a similar problem asking whether a decomposable regular language is decomposable via regular languages has a simple affirmative answer. In other words, if we know that a regular language L can be expressed as a product $L = X \cdot Y$, with $X, Y \neq \{1\}$, then X and Y can be replaced by regular languages, in fact even by regular supersets of X and Y, respectively. This fact, noticed already in [Co], is a simple consequence of the finiteness of the syntactic monoids of regular languages.

Some solutions of an equation (2) are easy to give: For a given X the sets like

$$Y = X^2, \quad Y = X + X^3 + X^7, \quad \text{and} \quad Y = X^+$$

are clearly solutions, while the set $X = A^+$ may or may not be a solution, cf. for example sets $X_1 = \{a, aaa, b, ab, ba, aba\}$ or $X_2 = \{a, ab, ba, bb\}$. Indeed, $\{a, b\}$ commutes with X_1 and b is not in any set commuting with X_2, since $ba \notin X_2A^+$.

Motivated, by the above examples, we can pose the following interesting problem:

Problem 6 (BTC-property). Are the following conditions equivalent:

(i) $XY = YX$;
(ii) there exist a set $V \subseteq A^+$ and sets of indices $I, J \subseteq \mathbb{N}$ such that

$$X = \bigcup_{i \in I} V^i \quad \text{and} \quad Y = \bigcup_{i \in J} V^j. \tag{3}$$

Problem 6 deserves several comments. First, the implication (ii) $\Rightarrow$ (i) is trivial. Second, the problem asks whether languages have a "similar" characterization for the commutation than the words have. Third, and most interestingly, a deep result of Bergman, cf. [Be], shows that the statement of the problem holds for polynomials over a field with noncommuting unknowns. It follows that Problem 6 has an affirmative solution for *finite multisets of words*. Amazingly

no combinatorial proof seems to be known for this nice result. Fourth, the abbreviation comes from Bergman's result: *Bergman type of characterization.*

Finally, we note that, in general, the answer to the question of Problem 6 is "no", a counterexample being X_1 above, i.e., the set $\{a, aaa, ab, b, ba, aba\}$, which commutes also with a finite set $X_3 = X_1 \cup \{aa\}$, and clearly X_1 and X_3 are not expressible in the form (3). Consequently, it remains a problem of finding families of languages for which the BTC-property hold, that is, for which the commutation of two sets can be characterized essentially as in the word case.

In what follows we give an overview of results related to the above problems. We start with a simple result, cf. [KP]:

Theorem 1. *The centralizer of a regular (or even recursive) language is in co-RE.*

Theorem 1 is achieved by a straightforward application of the exhaustive search principle. It immediately implies that Problem 5 can be stated equivalently as a problem whether the centralizer is recursively enumerable.

Concerning the partial solutions of Conway's Problem we have:

Theorem 2. *Conway's Problem has an affirmative answer in the following cases:*

(i) X *is a prefix code, see [Ra];*
(ii) $card(X) = 2$*, see [CKO];*
(iii) X *is a code which is elementary, synchronizing or contains a word of length* 1*, see [CKO];*
(iv) $card(X) = 3$*, see [KP];*
(v) X *is an* ω*-code, see [HP].*

Some of the above results are special cases of the others. All of them are listed here partially to point out the chronological order, but also to recall the original papers, since the proofs are typically quite different. All results except (v) are based on the combinatorial considerations on words, and in particular so-called Graph Lemma, see [ChK], plays an important role. The proof of (iv) is quite involved. Finally, the proof of (v) uses the above mentioned Bergman's theorem.

As a conclusion, Conway's Problem remains unanswered in many cases. Maybe the three most interesting subcases are the following:

- $card(X) < \infty$;
- $card(X) = 4$; or
- X is a finite code.

Typically, the proofs of results in Theorem 2 settle the validity of the BTC-property for the corresponding sets:

Theorem 3. *The BTC-property holds in the following cases:*

(i) X *is a prefix code, see [Ra];*

(ii) $card(X) = 2$, *see [CKO];*
(iii) X *is a code which is elementary, synchronizing or contains a word of length* 1, *see [CKO];*
(iv) $card(X) = 3$ *and* X *is a code, see [KP];*
(v) X *is an* ω*-code, see [HP].*

The only place where the conditions (i)-(v) are different in Theorems 2 and 3 is the case (iv): in Theorem 3 an additional requirement that X is a code is needed.

As we already mentioned the BTC-property does not hold even for all finite sets. A particularly small counterexample was given in [CKO]:

Example 1. Let $X = \{a, ab, ba, bb\}$. Then X commutes with

$$Y = X \cup X^2 \cup \{bab, bbb\}.$$

Clearly, X and Y do not satisfy the condition of (2). Note also that $\mathcal{C}(X) = \{a, b\}^+ \setminus \{b\}$.

From the above we can formulate the main open cases of Problem 6 as follows:

- $card(X) = 3$;
- $card(X) = 4$ and X is a code;
- $card(X) < \infty$ and X is a code.

There are no difficulties (except the computational ones) to test whether two regular languages commute. However, if we go into a bit more complex languages, the situation changes drastically, as shown by the following result proved in [HIKS].

Theorem 4. *Given a two-element set* $X = \{\alpha, \beta\}$, *with* $\alpha\beta \neq \beta\alpha$, *and a context-free language* L, *it is undecidable whether* X *and* L *commute.*

We proceed by illustrating a basic property of the centralizer of X (or in fact of any Y commuting with X). Clearly, we have:

$$X\mathcal{C}(X) = \mathcal{C}(X)X \Rightarrow X^n\mathcal{C}(X) = \mathcal{C}(X)X^n \text{ for all } n \geq 1.$$

Consequently, by König's Lemma, for any $z \in \mathcal{C}(X)$ and any sequence x_1, x_2, ... of elements of X we can find a sequence y_1, y_2, ... of elements of X such that $zx_1x_2\ldots = y_1y_2\ldots$. Similarly, we can consider extensions to left. The illustration is shown in Figure 1.

The property illustrated in Figure 1 has some nice consequences:

Fact 1. *For each* $X \subseteq A^+$ *we have*

$$X^+ \subseteq \mathcal{C}(X) \subseteq \mathrm{Pref}\,(X^+) \cap \mathrm{Suf}\,(X^+).$$

In order to state another fact, let us say that a set $X \subseteq A^+$ is *left* (resp. *right*) *complete* if, for each $w \in A^+$, $w \in Suf(X^+)$ (resp. $w \in Pref(X^+)$).

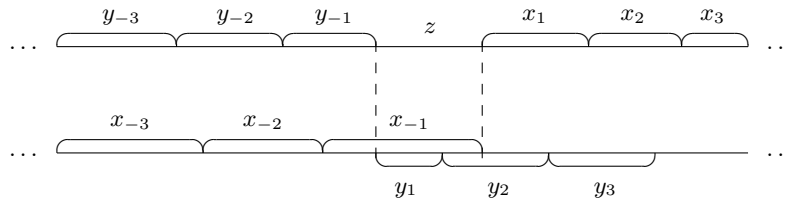


Fig. 1. An illustration how each $z \in \mathcal{C}(X)$ defines a double factorization of a bi-infinite word $\ldots y_{-3}y_{-2}y_{-1}y_1y_2y_3 \ldots = \ldots x_{-3}x_{-2}x_{-1}x_1x_2x_3 \ldots$.

Fact 2. *If $\mathcal{C}(X) = A^+$, then X is both left and right complete.*

Several simple observations can be based on Fact 2. The centralizer of a three element set X is never A^+, unless $X^+ = A^+$. Indeed, the only three-element sets which are both left and right complete are the sets of form $\{a, b, x\}$.

Further, by simple analysis only left and right complete four element sets are $X_1 = \{a, b, x, y\}$, $X_2 = \{a, ab, ba, bb\}$ and $X_3 = \{aa, ab, ba, bb\} = \{a, b\}^2$ (and the symmetric one to X_2). Clearly $X_1^+ = X_3^+ = A^+$, and as we already saw, $\mathcal{C}(X_2) = A^+ \setminus \{b\}$. Consequently, the centralizer of a four element set can not equal to A^+ either, as long as $X^+ \neq A^+$.

We conclude our considerations on the commutation with the following remark. So far we considered only languages not containing the empty word. Consequently, the centralizer was defined as the maximal semigroup commuting with a set. Of course, we can define it also as the maximal monoid commuting with a given set (which was actually the original formulation of Conway). There is no obvious connections between these problems, as discussed more in [CKO]. For example, the "monoid centralizer" of our set $X = \{a, ab, ba, bb\}$ in Example 1 is $\{a, b\}^*$ which is not equal modulo the empty word to the "semigroup centralizer" $\{a, b\}^+ \setminus \{b\}$. Note also that the "monoid centralizer" of a set $X \subseteq A^*$ containing the empty word is always A^*.

5.2 The Conjugacy

Now, the equation is

$$XZ = ZY, \tag{4}$$

and we look for its solutions in the family of finite languages. Setting $X = Y$ we are in the commutation problem, so that, due to the results of the previous section, there is no hope to have a full characterization of solutions of (4). We also note that considerations of Fig.1 are valid also for the conjugacy - factors denoted by x's and y's are from the sets Y and X, respectively. We recall that X and Y are *conjugates* if they satisfy (4) for some Z, and if this is the case, we

say that X and Y are *conjugated via* Z. These cases are denoted by $X \sim Y$ and $X \sim_Z Y$, respectively.

There are a number of interesting problems even in the case when only finite languages are considered. From the equations point of view we can ask:

Problem 7. Given a finite $Z \subseteq A^+$, characterize all finite sets $X, Y \subseteq A^+$ such that $X \sim_Z Y$.

Problem 8. Given finite sets $X, Y \subseteq A^+$, characterize all finite sets $Z \subseteq A^+$ such that $X \sim_Z Y$.

As a decision question it is natural to ask:

Problem 9. Decide whether two finite sets $X, Y \subseteq A^+$ are conjugated, i.e., whether there exists a set Z such that $X \sim_Z Y$.

We do not know whether it would make any difference if Z in Problem 9 would be required to be finite.

In what follows, we state what we know about these problems. The results are from [CKM], and they concentrate to the cases where either $card(Z) = 2$ or $card(X) = card(Y) = 2$.

To start with, we recall that (4) always has solutions of form

$$X = PQ,\ Y = QP, \text{ and } Z = P(QP)^i$$

for some (finite) sets P and Q and some integer $i \geq 0$. In fact, Z can be also of the form $Z = \bigcup_{i \in I} (PQ)^i P$. These are the solutions of *word-type* of the equation (4).

Example 2. Let $P, Q', P \subseteq A^+$ and $Z = \bigcup_{i \in I} (PQ)^i P$ for some $I \subseteq \mathbb{N}$, and moreover

$$PQ' \subseteq X \subseteq PQ \text{ and } QP \subseteq Y \subseteq Q'P.$$

Then, as is straightforward to see, P and Q are conjugated via Z, i.e., the equation can have also other types of solution than word-type ones. A concrete example is obtained by choosing $Q = \{1, b, bb\}$, $Q' = \{1, bb\}$, and $Z = P = \{a, ba\}$. Then $QP = Q'P$ and

$$PQ' = \{a, abb, ba, babb\} \subset \{a, ab, abb, ba, bab, babb\} = PQ,$$

so that by choosing

$$X = \{a, ab, abb, ba, babb\}, Y = \{a, abb, ba, babb\}$$

we see that

$$X \sim_Z Y.$$

Note that in this example $card(X) \neq card(Y)$, even if the set Z is almost the simplest possible: its cardinality is two and it is a prefix set!

The complications of Example 2 cannot occur if we restrict Z to be a biprefix set:

Fact 3. *If Z is a biprefix set and $X \sim_Z Y$, then necessarily $card(X) = card(Y)$.*

From now on we consider only the cases where either $card(Z) = 2$ or $card(X) = card(Y) = 2$. We call these cases *binary*. Even in these simple cases we do not have complete characterizations of Problems 7 and 8. However, based on Fact 3 and some more detailed related considerations (for more details see [CKM]) we have a complete answer to Problem 7 in the case when Z is a binary biprefix set.

Theorem 5. *Let $X, Y \subseteq A^+$ be finite and $Z \subseteq A^+$ a biprefix set with $card(Z) = 2$. Then $X \sim_Z Y$ if and only if X and Y can be partitioned into sets*

$$X = \bigcup_{i=1}^{t} X_i \text{ and } Y = \bigcup_{i=1}^{t} Y_i$$

with $card(X_i) = card(Y_i) = 2$, for $i = 1, \ldots, t$, such that

$$X_i \sim_Z Y_i \text{ for } i = 1, \ldots, t. \tag{5}$$

To complete the above characterization we have to deal with the relations in (5).

Theorem 6. *Let $X = \{x_1, x_2\}$ and $Y = \{y_1, y_2\}$ be binary sets and $Z = \{z_1, z_2\}$ a binary biprefix code. Then*

$$X \sim_Z Y$$

if and only if there exists words p_1, q_1, p_2 and q_2 and integers m_1 and m_2 such that

$$x_1 = p_1q_1,\ y_1 = q_1p_1,\ z = (p_1q_1)^{m_1}p_1$$
$$x_2 = p_2q_2,\ y_2 = q_2p_2,\ z = (p_2q_2)^{m_2}p_2,$$

and moreover,

$$q_1(p_2q_2)^{m_2} = (q_1p_1)^{m_1}q_2 \text{ and } q_2(p_1q_1)^{m_1} = (q_2p_2)^{m_2}p_1.$$

Now, we turn to Problem 8 in the binary case, that is in the case when both X and Y are binary. Our next result shows that the conjugacy of X and Y reduces to that of words. It provides a complete characterization, which, however, is not easy to state in a closed form. We denote by $\rho(w)$ the primitive root of w.

Theorem 7. *Let $X = \{x_1, x_2\} \subseteq A^+$ and $Y = \{y_1, y_2\} \subseteq A^+$ be conjugates, i.e., $X \sim Y$. Then at least one of the following conditions holds true:*

(i) $x_1x_2 = x_2x_1$, $y_1y_2 = y_2y_1$, *and* $\rho(x_1) \sim \rho(y_1)$;
(ii) $|x_1| = |x_2|$ *and there exists a word z such that, for $i = 1$ or 2, $x_1 \sim_z y_i$ and either $x_2 \sim_z y_{3-i}$ or $y_{3-i} \sim_z x_2$;*
(iii) $|x_1| \neq |x_2|$ *and $x_1 \sim y_i$ and $x_2 \sim y_{3-i}$ for $i = 1$ or 2.*

Note that the notation $x \sim_z y$ means that $xz = zy$, and therefore not necessarily implies that $y \sim_z x$. As we said it is not easy to see directly from Theorem 7 what is the characterization for the conjugacy of two element sets. However, it is obtainable via Proposition 2.

We conclude this section with two remarks on Problem 9. First of all we do not have a solution for it. In fact, we believe that it might contain similar intriguing features as Conway's Problem. However, we do know that it is decidable in the case of biprefix codes.

Theorem 8. *It is decidable whether two finite biprefix codes are conjugates.*

5.3 The Equivalence of Finite Substitutions on Regular Languages

In this final section we concentrate to the following decision problem:

Problem 10 (The equivalence of finite substitutions on regular language). Given a regular language $L \subseteq A^*$ and two finite substitutions $\varphi, \psi : A^* \to \mathcal{P}(B^*)$, decide whether or not φ and ψ are equivalent on L, i.e.,

$$\varphi(w) = \psi(w) \text{ for all } w \in L? \tag{6}$$

It is not difficult to see that this problem can be translated into an equivalence problem of finite transducers, the translation being based on the fact that the set of accepting computations of a finite automaton is regular as well, see [CuK]. More precisely, Problem 10 is equivalent to the decision problem asking to decide whether two input deterministic gsm's are equivalent. Here the *input determinism* means that the automaton is deterministic with respect to the input tape.

This translation emphasizes the importance of Problem 10, and in particular its surprising solution. In [Li], see also [HH], Problem 10 was shown to be undecidable even for a fixed simple language L.

Theorem 9. *It is undecidable whether two finite substitutions φ and ψ are equivalent on the language $L = ¢\{a, b\}^* £$.*

The structure of L in Theorem 9 motivates to state a simpler form of Problem 10:

Problem 11. Is it decidable whether two finite substitutions $\varphi, \psi : \{¢, a, £\}^* \to \mathcal{P}(B^*)$ are equivalent on the language $L' = ¢a^* £$.

The question in Problem 11 is very simple – but apparently amazingly difficult. In order to support this view we recall the following. First, as shown in [La], there does not exist any finite test set for L', i.e., a finite subset F of L' such that to test the equivalence of any two φ and ψ on L' it would be enough to do it on F. Second, some variants of Problem 11 are decidable, and some others are undecidable. Decidable variants are obtained by taking some special types of finite substitutions, see [KLI], for example, if the substitution φ is so-called

prefix substitution, i.e., for each letter c the set $\varphi(c)$ is a prefix set. Actually, in this setting the language can be any regular language, and moreover, instead of the equality (6) we can ask the inclusion:

$$\varphi(w) \subseteq \psi(w) \text{ for all } w \in L. \tag{7}$$

On the other hand, if we ask the inclusion, i.e., the validity of (7), for arbitrary finite substitutions the problem becomes undecidable even for the language L' :

Theorem 10. *The inclusion problem for finite substitutions φ and ψ on the language $L' = \not\in A^* \not\$$ is undecidable.*

This surprising result was first proved in [Tu] and later reproved in [KLII] in a slightly stronger form, that is in the form where $\varphi(c)$ is of cardinality 2 for each letter c. This extension yields also the following very simply formulated undecidability result:

Theorem 11. *Let $X = \{\alpha, \beta\} \subseteq A^+$ be binary, and let Y, $Z \subseteq A^+$ be finite. It is undecidable whether or not the inclusion*

$$X^n \subseteq YZ^{n-1}$$

holds for all $n \geq 1$.

Theorems 10 and 11, and in particular their proofs, reveal surprising facts about the computational power of finite sets. For example, the question of Theorem 11 can be interpreted as a two player game. The first player have two choices (α or β) and he chooses an arbitrary sequence of those, i.e., an arbitrary path in the complete binary tree. The second player has also a simple strategy: he first chooses an element from a finite set Y and then consecutively elements from the other finite set Z. The question is whether the second player can always respond with a sequence being equally long as that one chosen by the first player. Theorem 11 asserts that, in general, there is no algorithm to decide whether the second player always succeeds (to create the same word).

Despite of Theorems 10 and 11 our Problem 11 remains still unanswered.

Acknowledgement

The author is grateful to M. Hirvensalo and J. Manuch for the help in preparing this paper.

References

AL. Albert, M. H. and Lawrence, J., *A proof of Ehrenfeucht's Conjecture*, Theoret. Comput. Sci 41, pp. 121–123, 1985.

Be. Bergman, G. *Centralizers in free associative algebras*, Transactions of the American Mathematical Society 137, pp. 327–344, 1969.

CKM. Cassaigne, J., Karhumäki, J., and Manuch, J., *On conjugacy of languages*, manuscript in preparation.

CR. Crochemore, M. and Rytter, W., *Text Algorithms*, Oxford University Press, 1994.

ChK. Choffrut, C. and Karhumäki, J., *Combinatorics on Words.* In G. Rozenberg, A. Salomaa (eds.) *Handbook of Formal Languages*, vol 1, pp. 329–438, Springer-Verlag 1997.

CKO. Choffrut, C., Karhumäki, J., and Ollinger, N., *The Commutation of finite sets: a challenging problem.* Theoret. Comput. Sci., to appear.

Co. Conway, J., *Regular Algebra and Finite Machines*, Chapman Hall, 1971.

CuK. Culik II, K. and Karhumäki, J., *the equivalence of finite valued transducers (on HDT0L languages) is decidable*, Theoret. Comput. Sci 47, pp.71–84, 1986.

Di. Diekert, V., *Makanin's Algorithm* in M. Lothaire, *Algebraic Combinatorics on Words*, Cambridge University Press, to appear.

Gu. Guba, V. S., *The equivalence of infinite systems of equations in free groups and semigroups with finite systems* (in Russian), Mat. Zametki 40, pp. 321–324, 1986.

Ma. Makanin, S. G., *The problem of solvability of equation in a free semigroup*, Mat. Sb. 103, pp. 147–236, 1977 (English transl. in Math USSR Sb. 32, pp. 129–198).

HH. Halava, V. and Harju, T., *Undecidability of the equivalence of finite substitutions on regular language*, Theoret. Inform. and Appl. 33, pp. 117–124, 1999.

HIKS. Harju, T., Ibarra, O., Karhumäki, J. and Salomaa, A., *Decision questions concerning semilinearity, morphisms and commutation of languages*, TUCS report 376, 2000.

HP. Harju, T., Petre, I., personal communication.

HU. Hopcroft, J. and Ullman, J. D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA., 1979.

Ka. Karhumäki, J., *Some open problems on combinatorics of words and related areas, Proceedings of Algebraic Systems, Formal Languages and Computations*, RIMS Proceedings 1166, Research Institute for Mathematical Sciences, Kyoto University, pp. 118–130, 2000.

KLI. Karhumäki, J. and Lisovik, L.. *On the equivalence of finite substitutions and transducers*, In J. Karhumäki, H. Maurer, Gh. Păun and G. Rozenberg (eds.) *Jewels are Forever*, pp. 97–108, Springer 1999.

KLII. Karhumäki, J. and Lisovik, L., *A simple undecidable problem: The inclusion problem for finite substitutions on ab^*c*, Proceedings of STACS01, Lecture Notes in Computer Science, to appear.

KMP. Karhumäki, J., Mignosi, F., and Plandowski, W., *On expressibility of languages and relations by word equations*, JACM 47, 2000.

KP. Karhumäki, J. and Petre, I., *On the centralizer of a finite set*, Proceedings of ICALP 00, Lecture Notes in Computer Science 1853, 536–546, 2000.

La. Lawrence, J., *The non-existence of finite test sets for set equivalence of finite substitutions*, Bull. EATCS 28, pp. 34–37, 1986.

Li. Lisovik, L. P., *The equivalence problem for finite substitutions on regular languages*, Doklady of Academy of Sciences of Russia 357, pp. 299–301, 1997.

Lo. Lothaire, M., *Combinatorics on Words*, Adison-Wesley, Reading, MA., 1983.

Pl. Plandowski, W., *Satisfiability of word equations with constants is in PSPACE*, proceedings of FOCS'99, pp. 495–500, 1999.

Ra. Ratoandramanana, B., *Codes et motifs*, RAIRO Inform. Theor., 23:4, pp. 425–444, 1989.

Tu. Turakainen, P., *On some transducer equivalence problems for families of languages*, Intern. J. Computer Math. 23, 99–124, 1988.

Computing with Membranes (P Systems): Universality Results

Carlos Martín-Vide[1] and Gheorghe Păun[2]

[1] Research Group in Mathematical Linguistics
Rovira i Virgili University
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
cmv@astor.urv.es
[2] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 70700 Bucureşti, Romania
gpaun@imar.ro, g_paun@hotmail.com

Abstract. This is a survey of universality results in the area of Membrane Computing (P systems), at the level of December 2000[1]. We consider both P systems with symbol-objects and with string-objects; in the latter case, we consider systems based on rewriting, splicing, as well as rewriting together with other operations (replication, crossingover), with sets or with multisets of strings. Besides recalling characterizations of recursively enumerable languages and of recursively enumerable sets of vectors of natural numbers, we also briefly discuss the techniques used in the proofs of such results. Several open problems are also formulated.

1 Introduction; The Basic Idea

The P systems (initially, in [28], they were called *super-cell* systems) were introduced as a possible answer to the question whether or not the frequent statements (see, e.g., [3], [19]) that the processes which take place in a living cell are "computations", that "the alive cells are computers", are just metaphors, or a formal computing device can be abstracted from the cell functioning. As we will see below, the answer turned out to be affirmative.

Three are the fundamental features of alive cells which are basic to P systems: (1) the complex compartmentation by means of a **membrane structure**, where (2) **multisets** of chemical compounds evolve according to prescribed (3) **rules**.

A *membrane structure* is a hierarchical arrangement of membranes, all of them placed in a main membrane, called the *skin* membrane. This one delimits the system from its environment. The membranes should be understood as three-dimensional vesicles, but a suggestive pictorial representation is by means of planar Euler-Venn diagrams (see Figure 1). Each membrane precisely identifies a *region*, the space between it and all the directly inner membranes, if any exists. A membrane without any membrane inside is said to be *elementary*.

[1] An up-to-date bibliography of the area can be found at the web address http://bioinformatics.bio.disco.unimib.it/psystems

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 82–101, 2001.

In the regions of a membrane structure we place sets or *multisets* of *objects*. A multiset is a usual set with multiplicities associated with its elements, in the form of natural numbers; the meaning is that each object can appear in a number of identical copies in a given region. For the beginning, the objects are supposed to be symbols from a given alphabet (we will work with finitely many types of objects, that is, with multisets over a finite support-set), but later we will also consider string-objects.

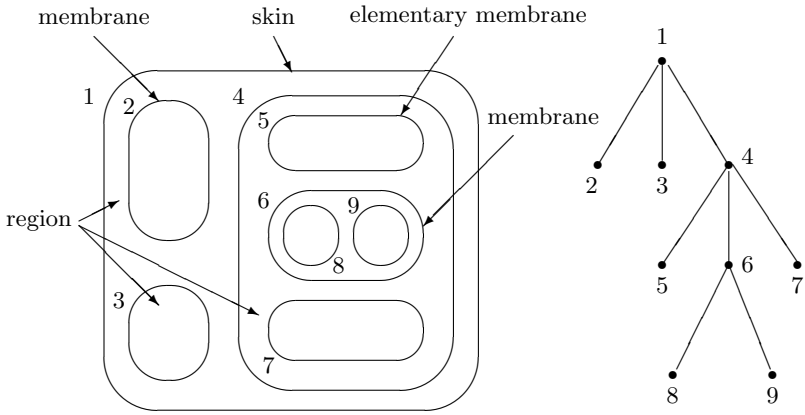


Fig. 1. A membrane structure and its associated tree

The objects evolve by means of given *rules*, which are associated with the regions (the intuition is that each region has specific chemical reaction conditions, hence the rules from a region cannot necessarily act also elsewhere). These rules specify both object transformation and object transfer from a region to another one. The passing of an object through a membrane is called *communication*.

Here is a typical rule:

$$cabb \to caad_{out}d_{in_3},$$

with the following meaning: one copy of the *catalyst* c (note that it is reproduced after the "reaction") together with one copy of object a and two copies of object b react together and produce one copy of c, two of a, and two copies of object d; one of these latter objects is sent out of the region where the rule is applied, while the second copy is sent to the adjacently inner membrane with the label 3, if such a membrane exists; the objects c, a, a remain in the same membrane (it is supposed that they have associated the communication command *here*, but we do not explicitly write this indication); if there is no membrane with label 3 directly inside the membrane where the rule is to be applied, then the rule cannot be applied. By a command *out*, an object can be also sent out of the skin membrane, hence it leaves the system and never comes back.

Therefore, the rules perform a multiset processing; in the previous case, the multiset represented by *cabb* is subtracted from the multiset of objects in a given region, objects *caa* are added to the multiset in that region, while copies of d are added to the multisets in the upper and lower regions.

The rules are used in a *nondeterministic maximally parallel manner*: the objects to evolve and the rules to be applied to them are chosen in a nondeterministic manner, but after assigning objects to rules no further rule should be applicable to the remaining objects. Sometimes, a priority relation among rules is considered, hence the rules to be used and the objects to be processed are selected in such a way that only rules which have a maximal priority among the applicable rules are used.

Other features can be considered, such as the possibility to control the membrane thickness/permeability, but we will introduce them later.

The membrane structure together with the multisets of objects and the sets of evolution rules present in its regions constitute a *P system*. The membrane structure and the objects define a *configuration* of a given P system. By using the rules as suggested above, we can define *transitions* among configurations. A sequence of transitions is called a *computation*. We accept as successful computations only the *halting* ones, those which reach a configuration where no further rule can be applied.

With a successful computation we can associate a *result*, for instance, by counting the multiplicity of objects which have left the system during the computation. More precisely, we can use a P system for solving three types of tasks: as a *generative* device (start from an initial configuration and collect all vectors of natural numbers describing the multiplicities of objects which have left the system during all successful computations), as a *computing* device (start with some input placed in an initial configuration and read the output at the end of a successful computation, by considering the objects which have left the system), and as a *decidability* device (introduce a problem in an initial configuration and wait for the answer in a specified number of steps). Here we deal only with the first case. Many classes of P systems turn out to be computationally universal, able to generate exactly what Turing machines can recursively enumerate.

2 A More Formal Definition of a P System

A membrane structure can be mathematically represented by a tree, in the natural way, or by a string of matching parentheses. The tree of the structure in Figure 1 is given in the same figure, while the parenthetic representation of that structure is the following:

$$[_1\,[_2\,]_2\,[_3\,]_3\,[_4\,[_5\,]_5\,[_6\,[_8\,]_8\,[_9\,]_9\,]_6\,[_7\,]_7\,]_4\,]_1.$$

The tree representation makes possible considering various parameters, such as the *depth* of the membrane structure, and also suggests considering membrane structures of particular types (described by linear trees, star trees, etc).

A multiset over an alphabet $V = \{a_1, \ldots, a_n\}$ is a mapping μ from V to $\mathbf{N}$, the set of natural numbers, and it can be represented by any string $w \in V^*$ such that $\Psi_V(w) = (\mu(a_1), \ldots, \mu(a_n))$, where Ψ_V is the Parikh mapping associated with V. Operations with multisets are defined in the natural manner.

With these simple prerequisites, we can define a P system (of *degree* $m \geq 1$) as a construct

$$\Pi = (V, T, C, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m)),$$

where:

(i) V is an alphabet; its elements are called *objects*;
(ii) $T \subseteq V$ (the *output* alphabet);
(iii) $C \subseteq V, C \cap T = \emptyset$ (*catalysts*);
(iv) μ is a membrane structure consisting of m membranes, with the membranes and the regions labeled in a one-to-one manner with elements of a given set H; in this section we use the labels $1, 2, \ldots, m$;
(v) $w_i, 1 \leq i \leq m$, are strings representing multisets over V associated with the regions $1, 2, \ldots, m$ of μ;
(vi) $R_i, 1 \leq i \leq m$, are finite sets of *evolution rules* over V associated with the regions $1, 2, \ldots, m$ 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.
An evolution rule is a pair (u, v), which we will usually write in the form $u \to v$, where u is a string over V and $v = v'$ or $v = v'\delta$, where v' is a string over $\{a_{here},\ a_{out},\ a_{in_j} \mid a \in V, 1 \leq j \leq m\}$, and δ is a special symbol not in V. The length of u is called *the radius* of the rule $u \to v$.

When presenting the evolution rules, the indication "here" is in general omitted.

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 a radius greater than one are of the form $ca \to cv$, 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 $c \to v$ or $a \to v_1 c v_2$, for $c \in C$).

The $(m + 1)$-tuple $(\mu, w_1, \ldots, w_m)$ constitutes the *initial configuration* of Π. In general, any sequence $(\mu', w'_{i_1}, \ldots, w'_{i_k})$, with μ' a membrane structure obtained by removing from μ all membranes different from $i_1, \ldots, i_k$ (of course, the skin membrane is not removed), with w'_j strings over V, $1 \leq j \leq k$, and $\{i_1, \ldots, i_k\} \subseteq \{1, 2, \ldots, m\}$, is called a *configuration* of Π.

It should be noted the important detail that the membranes preserve the initial labeling in all subsequent configurations; in this way, the correspondence between membranes, multisets of objects, and sets of evolution rules is well specified by the subscripts of these elements.

For two configurations $C_1 = (\mu', w'_{i_1}, \ldots, w'_{i_k})$, $C_2 = (\mu'', w''_{j_1}, \ldots, w''_{j_l})$, of Π we write $C_1 \Longrightarrow C_2$, and we say that we have a *transition* from C_1 to C_2, if we can pass from C_1 to C_2 by using the evolution rules appearing in $R_{i_1}, \ldots, R_{i_k}$ in the following manner.

Consider a rule $u \to v$ in a set R_{i_t}. We look to the region of μ' associated with the membrane i_t. If the objects mentioned by u, with the multiplicities at least as large as specified by u, appear in w'_{i_t}, then these objects can evolve according to the rule $u \to v$. The rule can be used only if no rule of a higher priority exists in R_{i_t} and can be applied at the same time with $u \to v$. More precisely, we start to examine the rules in the decreasing order of their priority and assign objects to them. A rule can be used only when there are copies of the objects whose evolution it describes and which were not "consumed" by rules of a higher priority and, moreover, there is no rule of a higher priority, irrespective which objects it involves, which is applicable at the same step. Therefore, all objects to which a rule *can* be applied *must* be the subject of a rule application. All objects in u are "consumed" by using the rule $u \to v$.

The result of using the rule is determined by v. If an object appears in v in the form a_{here}, then it will remain in the same region i_t. If an object appears in v in the form a_{out}, then a will exit membrane i_t and will become an element of the region which surrounds membrane i_t. In this way, it is possible that an object leaves the system: if it goes outside the skin of the system, then it never comes back. If an object appears in the form a_{in_q}, then a will be added to the multiset from membrane q, providing that the rule $u \to v$ was used in the region adjacent to membrane q. If a_{in_q} appears in v and membrane q is not one of the membranes delimiting "from below" the region i_t, then the application of the rule is not allowed.

If the symbol δ appears in v, then membrane i_t is removed (we say *dissolved*) and at the same time the set of rules R_{i_t} (and its associated priority relation) is removed. The multiset from membrane i_t is added (in the sense of multisets union) to the multiset associated with the region which was directly external to membrane i_t. We do not allow the dissolving of the skin membrane, because this means that the whole "cell" is lost, we do no longer have a correct configuration of the system.

All these operations are performed in parallel, for all possible applicable rules $u \to v$, for all occurrences of multisets u in the regions associated with the rules, for all regions at the same time. No contradiction appears because of multiple membrane dissolving, or because simultaneous appearance of symbols of the form a_{out} and δ. If at the same step we have a_{in_i} outside a membrane i and δ inside this membrane, then, because of the simultaneity of performing these operations, again no contradiction appears: we assume that a is introduced in membrane i at the same time when it is dissolved, thus a will remain in the region surrounding membrane i; that is, from the point of view of a, the effect of a_{in_i} in the region outside membrane i and δ in membrane i is a_{here}.

A sequence of transitions between configurations of a given P system Π is called a *computation* with respect to Π. A computation is *successful* if and only if it halts, that is, there is no rule applicable to the objects present in the last configuration. The result of a successful computation is $\Psi_T(w)$, where w describes the multiset of objects from T which have been sent out of the system during the

computation. The set of such vectors $\Psi_T(w)$ is denoted by $Ps(\Pi)$ (from "Parikh set") and we say that it is *generated* by Π.

3 Universality Results for Systems with Symbol-Objects

In this section we recall some results about the generative power of some variants of P systems working with symbol-objects. In many cases, characterizations of recursively enumerable sets of vectors of natural numbers (their family is denoted by $PsRE$) are obtained.

3.1 Further Features Used in P Systems

Before giving these results, we will specify some further ingredients which can be used in a P system. They are in general introduced with the aim of obtaining more "realistic" systems. For instance, instead of the powerful command in_j, which indicates the target of the destination membrane, we can consider weaker communication commands. The weakest one is to add no label to *in*: if an object a_{in} is introduced in some region of a system, then a will go to any of the adjacent lower membranes, nondeterministically chosen; if no inner membrane exists, then a rule which introduces a_{in} cannot be used.

An intermediate possibility is to associate both with objects and membranes *electrical charges*, indicated by $+, -, 0$ (positive, negative, neutral). The charges of membranes are given in the initial configuration and are not changed during computations, the charge of objects are given by the evolution rules, in the form $a \to b^+ d^-$. A charged object will immediately go into one of the directly lower membranes of the opposite polarization, nondeterministically chosen, the neutral objects remain in the same region or will exit it, according to the commands *here, out* associated with them.

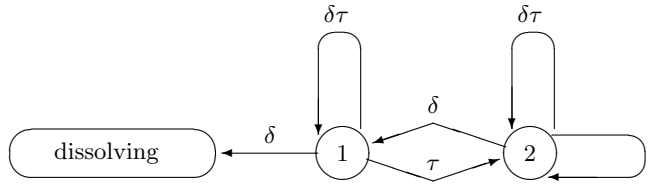


Fig. 2. The effect of actions δ, τ

Moreover, besides the action δ we can also consider an opposite action, denoted by τ, in order to control the membrane thickness (hence permeability). This is done as follows. Initially, all membranes are considered of thickness 1. If a rule in a membrane of thickness 1 introduces the symbol τ, then the membrane becomes of thickness 2. A membrane of thickness 2 does not become thicker by

using further rules which introduce the symbol τ, but no object can enter or exit it. If a rule which introduces the symbol δ is used in a membrane of thickness 1, then the membrane is dissolved; if the membrane had thickness 2, then it returns to thickness 1. If at the same step one uses rules which introduce both δ and τ in the same membrane, then the membrane does not change its thickness. These actions of the symbols δ, τ are illustrated in Figure 2.

No object can be communicated through a membrane of thickness two, hence rules which introduce commands *out, in*, requesting such communications, cannot be used. However, the communication has priority over changing the thickness: if at the same step an object should be communicated and a rule introduces the action τ, then the object is communicated and "afterthat" the membrane changes the thickness.

Also a variant of catalysts can be considered, with a "short term memory". Such catalysts (we call them *bi-stable*) have two states each, c and $\overline{c}$, and they can enter rules of the forms $ca \to \overline{c}v, \overline{c}a \to cv$ (always changing from c to $\overline{c}$ and back).

3.2 The Power of the Previous Systems

Consider now some notations. The family of sets of vectors of natural numbers $Ps(\Pi)$ generated by P systems with priority, catalysts, and the actions δ, τ, and of degree at most $m \geq 1$, using target indications of the form *here, out, in*, is denoted by $NP_m(Pri, Cat, i/o, \delta, \tau)$; when one of the features $\alpha \in \{Pri,\ Cat,\ \delta, \tau\}$ is not present, we replace it with $n\alpha$. We also write $2Cat$ instead of Cat when using bi-stable catalysts instead of usual catalysts.

Proofs of the following results can be found in [28], [12], [36]:

Theorem 1. $PsRE = NP_2(Pri, Cat, i/o, n\delta, n\tau) = NP_4(nPri, Cat, i/o, \delta, \tau) = NP_1(nPri, 2Cat,\ i/o, n\delta, n\tau)$.

It is an *open problem* whether or not systems of the type $(nPri, Cat, i/o, \delta, \tau)$ can characterize $PsRE$ when using less than four membranes.

3.3 Conditional Use of Rules

Starting from the observation that in the biochemistry of the cell certain reactions are enhanced/supressed by certain chemical compounds, such as enzymes, catalysts, hormons, in [1] one considers P systems with the rules applicability controlled by the contents of each region by means of certain *promoters* and *inhibitors*; these promoters/inhibitors are given as multisets of objects associated with given sets of rules. A rule from such a set (as usual, placed into a region) can be used in its region only if *all* the promoting objects are present, respectively, only if *not all* the inhibiting objects are present in that region.

Specifically, a *P system* (of degree $m \geq 1$), *with promoters* is a construct

$$\Pi = (V, \mu, w_1, \ldots, w_m, (R_{1,1}, p_{1,1}), \ldots, (R_{1,k_1}, p_{1,k_1}), \ldots, (R_{m,1}, p_{m,1}), \ldots, (R_{m,k_m}, p_{m,k_m})),$$

where all components are as usual in a P system (note that we use neither a terminal subset of V, nor catalysts) and $R_{i,1}, \ldots, R_{i,k_i}, k_i \geq 1$, are finite sets of *rules* present in region i of the system, while $p_{i,j}$ are strings over V, called *promoters*, $1 \leq i \leq m, 1 \leq j \leq k_i$. The rules are of the form $a \to v$, for $a \in V$, $v \in \{b_{tar} \mid b \in V, tar \in \{here, out, in\}\}^*$.

The rules are used as usual in P systems, in the maximally parallel manner, but *the rules from a set $R_{i,j}$ can be used only if the multiset represented by $p_{i,j}$ is present in region i*, for all $1 \leq i \leq m, 1 \leq j \leq k_i$.

An identical definition holds for systems with forbidding conditions, where the rules from $R_{i,j}$ are used only if the associated multiset $p_{i,j}$ is not included in the multiset of objects present in the region.

The maximum of $k_i, 1 \leq i \leq m$, is called the *promoter diversity* (resp., *inhibitor diversity*) of the system. The family of all sets $Ps(\Pi)$, computed as above by systems Π of degree at most $m \geq 1$ and with the promoter diversity not exceeding $k \geq 1$, is denoted by $PsP_m(i/o, prom_k)$. When using inhibitors, we replace $prom_k$ by $inhib_k$. We stress the fact that we do not use catalysts, priorities, or actions δ, τ, while the communication commands are of the form *here, out, in*. The following results are proved in [1]; they show a clear trade-off between the number of membranes and the promoter diversity of the used systems.

Theorem 2. $PsRE = PsP_6(i/o, prom_2) = PsP_4(i/o, prom_3) = PsP_3(i/o, prom_4) = PsP_1(i/o, prom_7) = PsP_3(i/o, inhib_6)$.

It is an *open problem* whether or not these results can be improved (for instance, using at the same time less membranes and a smaller promoter/inhibitor diversity).

4 P Systems with String-Objects

As we have mentioned in the Introduction, is also possible (this was considered already in [28]) to work with objects described by strings. The evolution rules should then be string processing rules, such as rewriting and splicing rules. As a result of a computation we can either consider the language of all strings computed by a system, or the number of strings produced by a system and sent out during a halting computation. In the first case one works with usual sets of strings (languages), not with multisets (each string is supposed to appear in exactly one copy), while in the latter case we have to work with multisets. We start by considering the set case.

4.1 Rewriting P Systems

We consider here the case of using string-objects processed by rewriting. Always we use only context-free rules, having associated target indications. Thus, the rules of our systems are of the form $(X \to v; tar)$, where $X \to v$ is a context-free rule over a given alphabet and $tar \in \{here, out, in\} \cup \{in_j \mid 1 \leq j \leq m\}$,

with the obvious meaning: the string produced by using this rule will go to the membrane indicated by *tar* (j is the label of a membrane). As above, we can also use priority relations among rules as well as the actions δ, τ, and then the rules are written in the form $(X \to x\alpha; tar)$, with $\alpha \in \{\delta, \tau\}$.

Formally, a *rewriting P system* is a construct

$$\Pi = (V, T, \mu, L_1, \ldots, L_m, (R_1, \rho_1), \ldots, (R_m, \rho_m)),$$

where V is an alphabet, $T \subseteq V$ (the terminal alphabet), μ is a membrane structure with m membranes labeled with $1, 2, \ldots, m$, $L_1, \ldots, L_m$ are finite languages over V (initial strings placed in the regions of μ), $R_1, \ldots, R_m$ are finite sets of context-free evolution rules, $\rho_1, \ldots, \rho_m$ are partial order relations over $R_1, \ldots, R_m$.

The language generated by Π is denoted by $L(\Pi)$ and it is defined as follows: we start from the initial configuration of the system and proceed iteratively, by transition steps performed by using the rules in parallel, to all strings which can be rewritten, obeying the priority relations; at each step, each string which can be rewritten must be rewritten, but this is done in a sequential manner, that is, only one rule is applied to each string; the actions δ, τ have the usual meaning; when the computation halts, we collect the terminal strings sent out of the system during the computation. We stress the fact that each string is processed by one rule only, the parallelism refers here to processing simultaneously all available strings by all applicable rules.

We denote by $RP_m(Pri, i/o, \delta, \tau)$ the family of languages generated by rewriting P systems of degree at most $m \geq 1$, using priorities, target indications of the form *here, out, in*, and actions δ, τ; as usual, we use *nPri*, $n\delta, n\tau$ when appropriate.

The following result is proved in [28] for the case of three membranes; the improvement to two membranes was given independently in [17] and [25].

Theorem 3. $RE = RP_2(Pri, i/o, n\delta, n\tau)$.

The powerful feature of using a priority relation can be avoided at the price of using membranes with a variable thickness. This was proved first in [39], [41], without a bound on the number of membranes, then the result has been improved in [12]:

Theorem 4. $RE = RP_4(nPri, i/o, \delta, \tau)$.

It is not known whether or not this result is optimal.

4.2 Splicing P Systems

The strings in a P system can also be processed by using the *splicing* operation introduced in [14] as a formal model of the DNA recombination under the influence of restriction enzymes and ligases (see a comprehensive information about splicing in [32]).

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 $u_1, u_2, u_3, u_4 \in V^*$ (V^* is the set of all strings over V). For such a rule r and for $x, y, w, z \in V^*$ we define

$$(x, y) \vdash_r (w, z) \text{ iff } x = x_1u_1u_2x_2,\ y = y_1u_3u_4y_2,\ w = x_1u_1u_4y_2,\ z = y_1u_3u_2x_2,$$
$$\text{for some } x_1, x_2, y_1, y_2 \in V^*.$$

(One cuts the strings x, y in between u_1, u_2 and u_3, u_4, respectively, and one recombines the fragments obtained in this way.)

A *splicing P system* (of degree $m \geq 1$) is a construct

$$\Pi = (V, T, \mu, L_1, \ldots, L_m, R_1, \ldots, R_m),$$

where V is an alphabet, $T \subseteq V$ (the *output* alphabet), μ is a membrane structure consisting of m membranes (labeled with $1, 2, \ldots, m$), $L_i, 1 \leq i \leq m$, are languages over V associated with the regions $1, 2, \ldots, m$ of μ, $R_i, 1 \leq i \leq m$, are finite sets of *evolution rules* associated with the regions $1, 2, \ldots, m$ of μ, given in the form $(r; tar_1, tar_2)$, where $r = u_1\#u_2\$u_3\#u_4$ is a usual splicing rule over V and $tar_1, tar_2 \in \{here, out, in\}$.

Note that, as usual in splicing systems, when a string is present in a region of our system, it is assumed to appear in arbitrarily many copies.

A transition in Π is defined by applying the splicing rules from each region of μ, in parallel, to all possible strings from the corresponding regions, and following the target indications associated with the rules. More specifically, if x, y are strings in region i and $(r = u_1\#u_2\$u_3\#u_4; tar_1, tar_2) \in R_i$ such that we can have $(x, y) \vdash_r (w, z)$, then w and z will go to the regions indicated by tar_1, tar_2, respectively. Note that after splicing, the strings x, y are still available in region i, because we have supposed that they appear in arbitrarily many copies (an arbitrarily large number of them were spliced, arbitrarily many remain), but if a string w, z, resulting from a splicing, is sent out of region i, then no copy of it remains here.

The result of a computation consists of all strings over T which are sent out of the system at any time during the computation. We denote by $L(\Pi)$ the language of all strings of this type. Note that in this subsection we do not consider halting computations, but we leave the process to continue forever and we just observe it from outside and collect the terminal strings leaving the system.

We denote by $SP_m(i/o)$ the family of languages $L(\Pi)$ generated by splicing P systems as above, of degree at most $m \geq 1$.

In [35] it was proved that $SP_3(i/o) = RE$; the result has been improved in [26] as follows.

Theorem 5. $RE = SP_2(i/o)$.

4.3 P Systems with Rewriting and Replication

Recently, in [18] a variant of rewriting P systems was considered, where rules of the form $X \to (u_1, tar_1)|| \ldots ||(u_n, tar_n)$ are used. When applying this rule to

a string, n strings are obtained, by replacing one occurrence of the symbol X by $u_1, \ldots, u_n$ and replicating the remaining part of the string; the n strings are sent to the membranes indicated by the targets $tar_1, \ldots, tar_n$, respectively. The definition of a replication rewriting system and of its generated language is as usual in P systems area.

The family of all languages $L(\Pi)$, computed as above by systems Π of degree at most $m \geq 1$ is denoted by $RRP_m(i/o)$.

The following result is from [20], where one solves the problem formulated as open in [18] whether or not the hierarchy on the number of membranes gives an infinite hierarchy (as expected, the answer is negative).

Theorem 6. $RE = RRP_6(i/o)$.

4.4 Rewriting P Systems with Conditional Communication

In this subsection, we recall the universality results from [2], where one considers a variant of rewriting P systems where the communication of strings is not controlled by the evolution rules, but it depends on the contents of the strings themselves. This is achieved by considering certain types of *permitting* and *forbidding* conditions, based on the symbols or the substrings (arbitrary, or prefixes/suffixes) which appear in a given string, or on the shape of the string.

First, let us mention that the set of symbols appearing in a string $x \in V^*$ is denoted by $alph(x)$ and the set of substrings of x is denoted by $Sub(x)$. A regular expression is said to be *elementary* if it has the star height at most one and uses the union at most for symbols in the alphabet.

A *rewriting P system* (of degree $m \geq 1$) with conditional communication is a construct

$$\Pi = (V, T, \mu, L_1, \ldots, L_m, R_1, P_1, F_1, \ldots, R_m, P_m, F_m),$$

where the components $V, T, \mu, L_1, \ldots, L_m$ are as usual in rewriting P systems, $R_1, \ldots, R_m$ are finite sets of context-free *rules* over V present in region i of the system, P_i are *permitting conditions* and F_i are *forbidding conditions* associated with region $i, 1 \leq i \leq m$.

The conditions can be of the following forms:

1. *empty*: no restriction is imposed on strings, they exit the current membrane or enter any of the directly inner membrane freely.
2. *symbols checking*: each P_i is a set of pairs $(a, \alpha), \alpha \in \{in, out\}$, for $a \in V$, and each F_i is a set of pairs $(b, not\alpha), \alpha \in \{in, out\}$, for $b \in V$; a string w can go to a lower membrane only if there is a pair $(a, in) \in P_i$ with $a \in alph(w)$, and for each $(b, notin) \in F_i$ we have $b \notin alph(w)$; similarly, for sending the string w out of membrane i it is necessary to have $a \in alph(w)$ for at least one pair $(a, out) \in P_i$ and $b \notin alph(w)$ for all $(b, notout) \in F_i$.
3. *substrings checking*: each P_i is a set of pairs $(u, \alpha), \alpha \in \{in, out\}$, for $u \in V^+$, and each F_i is a set of pairs $(v, not\alpha), \alpha \in \{in, out\}$, for $v \in V^+$; a string w can go to a lower membrane only if there is a pair $(u, in) \in P_i$ with

$u \in Sub(w)$, and for each $(v, notin) \in F_i$ we have $v \notin Sub(w)$; similarly, for sending the string w out of membrane i it is necessary to have $u \in Sub(w)$ for at least one pair $(u, out) \in P_i$ and $v \notin Sub(w)$ for all $(v, notout) \in F_i$.

4. *prefix/suffix checking*: exactly as in the case of substrings checking, with the checked string being a prefix or a suffix of the string to be communicated.
5. *shape checking*: each P_i is a set of pairs $(e, \alpha), \alpha \in \{in, out\}$, where e is an elementary regular expression over V, and each F_i is a set of pairs $(f, not\alpha), \alpha \in \{in, out\}$, where f is an elementary regular expression over V; a string w can go to a lower membrane only if there is a pair $(e, in) \in P_i$ with $w \in L(e)$, and for each pair $(f, notin) \in F_i$ we have $w \notin L(f)$; similarly, for sending the string w out of membrane i it is necessary to have $w \in L(e)$ for at least one pair $(e, out) \in P_i$ and $w \notin L(f)$ for all $(f, notout) \in F_i$.

We say that we have conditions of the types *empty, symb, sub$_k$, pref$_k$, suff$_k$, patt*, respectively, where k is the length of the longest string in all P_i, F_i.

The transitions in a system as above are defined in the usual way. In each region, each string which can be rewritten is rewritten by a rule from that region. Each string obtained in this way is checked against the conditions P_i, F_i from that region. If it fulfills the requested conditions, then it will be immediately sent out of the membrane or to an inner membrane, if any exists; if it fulfills both *in* and *out* conditions, then it is sent either out of the membrane or to a lower membrane, nondeterministically choosing the direction. If a string does not fulfill any condition, or it fulfills only *in* conditions and there is no inner membrane, then the string remains in the same region. A string which is rewritten and a string which is sent to another membrane is "consumed", we do not have a copy of it at the next step in the same membrane. If a string cannot be rewritten, then it is directly checked against the communication conditions, and, as above, it leaves the membrane (or remains inside forever) depending on the result of this checking.

That is, the rewriting has priority over communication: we first try to rewrite a string and only after that we try to communicate the result of the rewriting or the string itself if no rewriting is possible on it.

The family of all languages $L(\Pi)$, computed as above by systems Π of degree at most $m \geq 1$, with permitting conditions of type α, and forbidding conditions of type β, is denoted by $RP_m(\alpha, \beta)$, $\alpha, \beta \in \{$*empty, symb, patt*$\} \cup \{sub_k, pref_k,$ *suff*$_k \mid k \geq 1\}$. When we will use both prefix and suffix checking (each condition string can be checked both as a prefix or as a suffix, that is, we do not separately give sets of prefixes and sets of suffixes), then we indicate this by *prefsuff*$_k$. If the degree of the systems is not bounded, then the subscript m is replaced by $*$.

Proofs of the following results can be found in [2]. Again, a clear trade-off between the number of membranes and the power of the communication conditions is found. We do not know whether or not these results are optimal (for instance, as the number of used membranes); in particular, we do not have a proof of the fact that a reduced number of membranes suffice also when checking prefixes and suffixes, but we *conjecture* that such a result holds true.

Theorem 7. $RE = RP_2(patt, empty) = RP_2(empty, sub_2) = RP_4(sub_2, symb) = RP_6(symb, empty) = RP_*(prefsuff_2, empty)$.

4.5 P Systems with Leftmost Rewriting

Following [7], we now consider a restriction in the use of rules of a rewriting P system, of a language-theoretic nature: any string is rewritten in the leftmost position which can be rewritten by a rule from its region. That is, we examine the symbols of the string, step by step, from left to right; the first one which can be rewritten by a rule from the region of the string is rewritten. If there are several rules with the same left hand symbol, any one is chosen.

We denote by $L_{left}(\Pi)$ the language generated by a system Π in this way and by $RP_m(left), m \geq 1$, we denote the family of all such languages, generated by systems with at most m membranes. In view of the previous results, the following theorem is expected (but whether or not it is optimal in the number of membranes remains as an *open problem*):

Theorem 8. $RE = RP_6(left)$.

4.6 P Systems with Worm-Objects

In P systems with symbol-objects we work with multisets and the result of a computation is a vector of natural numbers; in the case of string-object P systems we work with sets of strings and the result of a computation is a string. We can combine the two ideas: we can work with multisets of strings and consider as the result of a computation the number of strings sent out during a halting computation. To this aim, we need operations with strings which can increase and decrease the number of occurrences of strings.

The following four operations were considered in [5] (they are slight variants of the operations used in [38]): *rewriting* (called *point mutation* in [5] and [38]), *replication* (as in Subsection 4.3, but always producing only two strings), *splitting* (if $a \in V$ and $u_1, u_2 \in V^+$, then $r : a \to u_1|u_2$ is called a *splitting rule* and we define the operation $x_1 a x_2 \Longrightarrow_r (x_1 u_1, u_2 x_2)$), *recombination/crossovering* (for a string $z \in V^+$ we define the operation $(x_1 z x_2, y_1 z y_2) \Longrightarrow_z (x_1 z y_2, y_1 z x_2)$).

Note that replication and splitting increase the number of strings, mutation and recombination not; by sending strings out of the system, their number can also be decreased.

A *P system* (of degree $m \geq 1$) *with worm-objects* is a construct

$$\Pi = (V, \mu, A_1, \ldots, A_m, (R_1, S_1, M_1, C_1), \ldots, (R_m, S_m, M_m, C_m)),$$

where:

- V is an alphabet;
- μ is a membrane structure of degree m (with the membranes labeled by $1, 2, \ldots, m$);

- $A_1, \ldots, A_m$ are multisets of a finite support over V^*, associated with the regions of μ;
- for each $1 \leq i \leq m$, R_i, S_i, M_i, C_i are finite sets of replication rules, splitting rules, mutation rules, and crossing-over blocks, respectively, given in the following forms:
  a. replication rules: $(a \to u_1||u_2; tar_1, tar_2)$, for $tar_1, tar_2 \in \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$;
  b. splitting rules: $(a \to u_1|u_2; tar_1, tar_2)$, for $tar_1, tar_2 \in \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$;
  c. mutation rules: $(a \to u; tar)$, for $tar \in \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$;
  d. crossing-over blocks: $(z; tar_1, tar_2)$, for $tar_1, tar_2 \in \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$.

The transitions are defined as usual in P systems area, according to the following specific rules: A string which enters an operation is "consumed" by that operation, its multiplicity is decreased by one. The multiplicity of strings produced by an operation is accordingly increased. A string is processed by only one operation. For instance, we cannot apply two mutation rules, or a mutation rule and a replication one, to the same string. The strings resulting from an operation are communicated to the region specified by the target indications associated with the used rule. (Note that when we have two resulting strings, two targets are associated with the rule.)

The result of a halting computation consists of the number of strings sent out of the system during the computation. A non-halting computation provides no output. For a system Π, we denote by $N(\Pi)$ the set of numbers computed in this way. By $NWP_m(tar), m \geq 1$, we denote the sets of numbers computed by all P systems with at most m membranes.

In [5] it is proved that each recursively enumerable set of natural numbers (their family is denoted by nRE) can be computed by a P system as above; the result is improved in [23], where it is shown that the hierarchy on the number of membranes collapses:

Theorem 9. $nRE = NWP_6(tar)$.

It is an *open problem* whether or not the bound 6 in this theorem can be improved; we expect a positive answer.

Note the resemblance of P systems with worm objects with P systems with rewriting and replication (also Theorems 6 and 9 are similar), but also the essential difference between them: in Subsection 4.3 we have used sets of strings, while here we have worked with multisets (and we have generated sets of vectors of natural numbers). Following the same strategy as in the case of using both rewriting and replication, we can consider also other combinations of operations from those used in the case of worm-objects. The case of rewriting and crossovering was investigated in [22]. The work of such a P system is exactly as the work of a P system with worm-objects, only the way of defining the result of a computation is different: we consider the language of all strings which leave the system during the halting computations.

Let us denote by $RXP_m(i/o), m \geq 1$, the family of languages generated by such systems with at most m membranes, using as communication commands the indications *here, out, in* (but not priorities and actions δ, τ). Somewhat expected, we get one further characterization of recursively enumerable languages (the proof can be found in [22]).

Theorem 10. $RE = RXP_5(i/o)$.

It is an *open problem* whether or not the bound 5 is optimal.

5 P Systems with Active Membranes

Let us now consider P systems where the membranes themselves are involved in rules. Such systems were introduced in [30] with also the possibility of dividing membranes, and their universaity was proved for the general case. However, in [25] it was proven that membrane division is not necessary. Here we will consider this restricted case. (However, membrane division is crucial in solving NP-complete problems in polynomial – even linear – time, by making use of an exponential working space created by membrane division, but we do not deal here with this aspect. A survey of results of this type can be found in [31].)

A *P system with active membranes, in the restricted form*, is a construct

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

where:

(i) $m \geq 1$;
(ii) V is the alphabet of the system;
(iii) $T \subseteq V$ (terminal alphabet);
(iv) H is a finite set of *labels* for membranes;
(v) μ is a *membrane structure*, consisting of m membranes labeled with elements of H and having a neutral charge (all membranes are marked with 0);
(vi) $w_1, \ldots, w_m$ are strings over V, describing the *multisets of objects* placed in the m regions of μ;
(vii) R is a finite set of *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) $a[_h]_h^\alpha \to [_h b]_h^\beta$, where $a, b \in V,\ h \in H,\ \alpha, \beta \in \{+, -, 0\}$ (an object is introduced in membrane h),
(c) $[_h a]_h^\alpha \to [_h]_h^\beta b$, for $h \in H,\ \alpha, \beta \in \{+, -, 0\},\ a, b \in V$ (an object is sent out of membrane h).

Also rules for dissolving and for dividing a membrane are considered in [30] (and in other subsequent papers), but we do not use such rules here.

The rules are used as customary in a P system, in a maximally parallel manner: in each time unit, all objects which can evolve, have to evolve. Each

copy of an object and each copy of a membrane can be used by only one rule, with the exception of rules of types (a), where we count only the involved object, not also the membrane. That is, if we have several objects a in a membrane i and a rule $[_i a \to v]_i^\alpha$, then we use this rule for all copies of a, irrespective how many they are; we do not consider that the membrane was used – note that its electrical charge is not changed. However, if we have a rule $[_i a]_i^\alpha \to [_i\,]_i^\beta b$, then this counts as using the membrane, no other rule of types (b) and (c) which involves the same membrane can be used at the same time.

As any other membrane, the skin membrane can be "electrically charged". During a computation, objects can leave the skin membrane (using rules of type (c)).

We denote by $N(\Pi)$ the set of all vectors of natural numbers computed as above by a P system Π. The family of all such sets of vectors, computed by systems with at most $m \geq 1$ membranes, is denoted by $PsAP_m$; when the number of membranes is not restricted, we replace the subscript m by $*$.

As announced above, in [25] it is proved that $PsRE = PsAP_*$ and the problem is formulated whether or not the hierarchy on the number of membranes collapses at a reasonable level. In [12] it is proved that four membranes suffice (it is an *open problem* whether or not this result is optimal).

Theorem 11. $PsRE = PsAP_4$.

6 Techniques Used in the Proofs of Universality Results

The most used tool (almost always used when dealing with P systems with symbol-objects) for proving universality results is the characterization of recursively enumerable languages by means of *matrix grammars with appearance checking* in the binary normal form. Such a grammar is a construct $G = (N, T, S, M, F)$, where N, T are disjoint alphabets, $S \in N$, M is a finite set of sequences of the form $(A_1 \to x_1, \ldots, A_n \to x_n)$, $n \geq 1$, of context-free rules over $N \cup T$ (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases), and F 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 $w \Longrightarrow z$ if there is a matrix $(A_1 \to x_1, \ldots, A_n \to x_n)$ 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 (1) $w_i = w'_i A_i w''_i, w_{i+1} = w'_i x_i w''_i$, for some $w'_i, w''_i \in (N \cup T)^*$, or (2) $w_i = w_{i+1}$, A_i does not appear in w_i, and the rule $A_i \to x_i$ appears in F. (The rules of a matrix are applied in order, possibly skipping the rules in F if they cannot be applied – we say that these rules are applied in the *appearance checking* mode.)

The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac}. When $F = \emptyset$ (hence we do not use the appearance checking feature), the generated family is denoted by MAT.

It is known that $CF \subset MAT \subset MAT_{ac} = RE$, the inclusions being proper.

A matrix grammar $G = (N, T, S, M, F)$ is said to be in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \#\}$, 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. $(X \to Y, A \to \#)$, 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 F consists exactly of all rules $A \to \#$ appearing in matrices of type 3; # is a trap-symbol, once introduced, it is never removed. A matrix of type 4 is used only once, in the last step of a derivation.

According to [6], for each matrix grammar there is an equivalent matrix grammar in the binary normal form.

For an arbitrary matrix grammar $G = (N, T, S, M, F)$, let us denote by $ac(G)$ the cardinality of the set $\{A \in N \mid A \to \alpha \in F\}$. ¿From the construction in the proof of Lemma 1.3.7 in [6] one can see that if we start from a matrix grammar G and we get the grammar G' in the binary normal form, then $ac(G') = ac(G)$.

Improving the result from [27] (six nonterminals, all of them used in the appearance checking mode, suffice in order to characterize RE with matrix grammars), in [13] it was proved that four nonterminals are sufficient in order to characterize RE by matrix grammars and out of them only three are used in appearance checking rules. Of interest in the P area is another result from [13]: if the total number of nonterminals is not restricted, then each recursively enumerable language can be generated by a matrix grammar G such that $ac(G) \leq 2$.

Consequently, to the properties of a grammar G in the binary normal form we can add the fact that $ac(G) \leq 2$. We will say that this is *the strong binary normal form* for matrix grammars.

Starting from such a grammar G, in many cases a P system is constructed with some membranes simulating the matrices from G which do not contain rules to be used in the appearance checking mode, and some membranes associated with the symbols which are checked in the matrices which contain rules used in the appearance checking mode (that is why the number of nonterminals A which appear in rules of the form $A \to \#$ is so important). This leads to a reduced number of membranes, as seen in the theorems from the previous sections.

Other very useful tools used mainly in the proofs dealing with string-objects are the *normal forms* of Chomsky grammars known to generate all recursively enumerable languages. The most important are the *Kuroda normal form* and the *Geffert normal form*.

A type-0 grammar $G = (N, T, S, P)$ is said to be in the *Kuroda normal form* if the rules from P are of one of the following two forms: $A \to x, AB \to CD$, for $A, B, C, D \in N$ and $x \in (N \cup T)^*$ (that is, besides context-free rules we have only rules which replace two nonterminals by two nonterminals).

A type-0 grammar $G = (N, T, S, P)$ is said to be in the *Geffert normal form* if $N = \{S, A, B, C\}$, and the rules from P are of one of the following two forms:

$S \to xSy$, with $x, y \in (T \cup \{A, B, C\})^*$, and $ABC \to \lambda$ (that is, besides context-free rules we have only one non-context-free rule, the erasing one $ABC \to \lambda$, with A, B, C being the only nonterminals of G different from S).

At a more technical level, the proofs of universality start by a matrix grammar or a grammar in Kuroda or Gefffert normal form, and construct a P system which simulates it; the control of the correct behavior of the system is ensured by various tricks, directly related to the type of the system we construct. In most cases, because only halting computations are accepted, a "wrong" choice of rules or of communication of objects is prevented by introducing trap-symbols which are able to evolve forever; in the string case, if a terminal alphabet is used, then one introduces trap-symbols which just prevent the string to become terminal. Another useful trick is to control the communication possibilities; if a rule introduces an object which must be communicated to a lower membrane, then the communication should be possible (for instance, the thickness of the lower membranes is not increased). A nice way to control the appearance of a symbol to be checked by a rule of a matrix grammar is provided by replication: a copy of the string is sent to a membrane where nothing happens except that the computation goes forever in the case when the specified symbol appears in the string; if this is not the case, then the other copy of the string will continue a "correct" development. When we have to work on strings in the leftmost/rightmost manner, then the rotate-and-simulate technique from the splicing area is useful: we simulate the rules of the starting grammar in the ends of the strings generated by a P system, and the strings are circularly permuted in order to make possible such a simulation in any place of a sentential form of the grammar.

Still more precise proof techniques (for instance, for synchronizing the work of different membranes) can be found in the literature, but we do not recall them here.

7 Final Remarks

We have considered most of the variants of P systems with symbol-objects and with string-objects and we have recalled characterizations of the family of recursively enumerable sets of vectors of natural numbers or of the family of recursively enumerable languages. Many other variants, several of them leading to similar results, can be found in the literature. We only mention here the generalized P systems considered in [8], [10], [11], the carrier-based systems (where no object evolves, but only passes through membranes) [24], the P systems with valuations [21] (their power is not know yet), the systems also able to create membranes [15], which lead to a characterization of Parikh sets of ET0L languages, and the systems also taking into account the energy created/consumed by each evolution rule [34].

Note. The work of the second author was supported by a grant of NATO Science Committee, Spain, 2000–2001.

References

1. P. Bottoni, C. Martin-Vide, Gh. Păun, G. Rozenberg, Membrane systems with promoters/inhibitors, submitted, 2000.
2. P. Bottoni, A. Labella, C. Martin-Vide, Gh. Păun, Rewriting P systems with conditional communication, submitted, 2000.
3. D. Bray, Protein molecules as computational elements in living cells, *Nature*, 376 (1995), 307–312
4. C. Calude, Gh. Păun, *Computing with Cells and Atoms*, Taylor and Francis, London, 2000.
5. J. Castellanos, A. Rodriguez-Paton, Gh. Păun, Computing with membranes: P systems with worm-objects, *IEEE 7th. Intern. Conf. on String Processing and Information Retrieval, SPIRE 2000*, La Coruna, Spain, 64–74.
6. J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.
7. Cl. Ferretti, G. Mauri, Gh. Păun, Cl. Zandron, Further remarks on P systems with string-objects, submitted, 2000.
8. R. Freund, Generalized P systems, *Fundamentals of Computation Theory, FCT'99*, Iaşi, 1999 (G. Ciobanu, Gh. Păun, eds.), *LNCS* 1684, Springer, 1999, 281–292.
9. R. Freund, Generalized P systems with splicing and cutting/recombination, *Workshop on Formal Languages*, FCT'99, Iaşi, 1999, *Grammars*, 2, 3 (1999), 189–199.
10. R. Freund, Sequential P systems, *Pre-proc. Workshop on Multiset Processing*, Curtea de Argeş, Romania, 2000, and *Theorietag 2000; Workshop on New Computing Paradigms* (R. Freund, ed.), TU University Vienna, 2000, 177–183.
11. R. Freund, F. Freund, Molecular computing with generalized homogeneous P systems, *Proc. Conf. DNA6* (A. Condon, G. Rozenberg, eds.), Leiden, 2000, 113–125.
12. R. Freund, C. Martin-Vide, Gh. Păun, Computing with membranes: Three more collapsing hierarchies, submitted, 2000.
13. R. Freund, Gh. Păun, On the number of non-terminals in graph-controlled, programmed, and matrix grammars, submitted, 2000.
14. T. Head, Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors, *Bulletin of Mathematical Biology*, 49 (1987), 737–759.
15. M. Ito, C. Martin-Vide, Gh. Păun, A characterization of Parikh sets of ET0L languages in terms of P systems, submitted, 2000.
16. 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.
17. S. N. Krishna, R. Rama, On the power of P systems with sequential and parallel rewriting, *Intern. J. Computer Math.*, 77, 1-2 (2000), 1–14.
18. S. N. Krishna, R. Rama, P systems with replicated rewriting, *J. Automata, Languages, Combinatorics*, to appear.
19. W. R. Loewenstein, *The Touchstone of Life. Molecular Information, Cell Communication, and the Foundations of Life*, Oxford Univ. Press, New York, 1999.
20. V. Manca, C. Martin-Vide, Gh. Păun, On the power of P systems with replicated rewriting, *J. Automata, Languages, Combinatorics*, to appear.
21. C. Martin-Vide, V. Mitrana, P systems with valuations, *Proc. Second Conf. Unconventional Models of Computing* (I. Antoniou, C. S. Calude, M. J. Dinneen, eds.), Springer-Verlag, 2000, 154–166.

22. C. Martin-Vide, Gh. Păun, String objects in P systems, *Proc. of Algebraic Systems, Formal Languages and Computations Workshop*, Kyoto, 2000, RIMS Kokyuroku, Kyoto Univ., 2000, 161–169.
23. C. Martin-Vide, Gh. Păun, Computing with membranes. One more collapsing hierarchy, *Bulletin of the EATCS*, 72 (2000), 183–187.
24. C. Martin-Vide, Gh. Păun, G. Rozenberg, Membrane systems with carriers, *Theoretical Compuer Sci.*, to appear.
25. A. Păun, On P systems with membrane division, *Proc. Second Conf. Unconventional Models of Computing* (I. Antoniou, C. S. Calude, M. J. Dinneen, eds.), Springer-Verlag, 2000, 187–201.
26. A. Păun, M. Păun, On the membrane computing based on splicing, in vol. *Words, Languages, Grammars* (C. Martin-Vide, V. Mitrana, eds.), Kluwer, Dordrecht, 2000, 409–422.
27. Gh. Păun, Six nonterminals are enough for generating each r.e. language by a matrix grammar, *Intern. J. Computer Math.*, 15 (1984), 23–37.
28. Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143 (see also *Turku Center for Computer Science-TUCS Report* No 208, 1998, www.tucs.fi).
29. Gh. Păun, Computing with membranes – A variant: P systems with polarized membranes, *Intern. J. of Foundations of Computer Science*, 11, 1 (2000), 167–182.
30. Gh. Păun, P systems with active membranes: Attacking NP-complete problems, *J. Automata, Languages and Combinatorics*, 6, 1 (2001).
31. Gh. Păun, Computing with membranes; Attacking NP-complete problems, *Proc. Second Conf. Unconventional Models of Computing* (I. Antoniou, C. S. Calude, M. J. Dinneen, eds.), Springer-Verlag, 2000, 94–115.
32. Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.
33. Gh. Păun, G. Rozenberg, A. Salomaa, Membrane computing with external output, *Fundamenta Informaticae*, 41, 3 (2000), 259–266,
34. Gh. Păun, Y. Suzuki, H. Tanaka, P Systems with energy accounting, *Intern. J. Computer Math.*, to appear.
35. 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.
36. Gh. Păun, S. Yu, On synchronization in P systems, *Fundamenta Informaticae*, 38, 4 (1999), 397–410.
37. G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, Springer-Verlag, Heidelberg, 1997.
38. M. Sipper, Studying Artificial Life using a simple, general cellular model, *Artificial Life Journal*, 2, 1 (1995), 1–35.
39. Cl. Zandron, G. Mauri, Cl. Ferretti, Universality and normal forms on membrane systems, *Proc. Intern. Workshop Grammar Systems 2000* (R. Freund, A. Kelemenova, eds.), Bad Ischl, Austria, July 2000, 61–74.
40. Cl. Zandron, Cl. Ferretti, G. Mauri, Solving NP-complete problems using P systems with active membranes, *Proc. Second Conf. Unconventional Models of Computing* (I. Antoniou, C. S. Calude, M. J. Dinneen, eds.), Springer-Verlag, 2000, 289–301.
41. Cl. Zandron, Cl. Ferretti, G. Mauri, Using membrane features in P systems, *Pre-proc. Workshop on Multiset Processing*, Curtea de Argeş, Romania, TR 140, CDMTCS, Univ. Auckland, 2000, 296–320.

A Simple Universal Logic Element and Cellular Automata for Reversible Computing

Kenichi Morita

Hiroshima University, Faculty of Engineering,
Higashi-Hiroshima, 739-8527, Japan
morita@ke.sys.hiroshima-u.ac.jp
http://www.ke.sys.hiroshima-u.ac.jp/~morita

Abstract. Reversible computing is a paradigm of computation that reflects physical reversibility, and is considered to be important when designing a logical devices based on microscopic physical law in the near future. In this paper, we focus on a problem how universal computers can be built from primitive elements with very simple reversible rules. We introduce a new reversible logic element called a "rotary element", and show that any reversible Turing machine can be realized as a circuit composed only of them. Such reversible circuits work in a very different fashion from conventional ones. We also discuss a simple reversible cellular automaton in which a rotary element can be implemented.

1 Introduction

Recently, various computing models that directly reflect laws of Nature have been proposed and investigated. They are quantum computing (e.g., [2,4]), DNA computing (e.g., [11]), reversible computing (e.g., [1,2,3]), and so on. Reversible computing is a model reflecting physical reversibility, and has been known to play an important role when studying inevitable power dissipation in a computing process. Until now, several reversible systems such as reversible Turing machines, reversible cellular automata, and reversible logic circuits have been investigated.

A logic gate is called reversible if its logical function is one-to-one. A reversible logic circuit is a one constructed only of reversible gates. There are "universal" reversible gates in the sense that any logic circuit (even if it is irreversible) can be embedded in a circuit composed only of them. A Fredkin gate [3] and a Toffoli gate [12] are typical universal reversible gates having 3 inputs and 3 outputs.

A rotary element (RE), which is also a reversible logic element, was introduced by Morita, Tojima, and Imai [8]. They proposed a simple model of 2-D reversible cellular automaton called P_4 in which any reversible two-counter machine can be embedded in a finite configuration. They showed that an RE and a position marker as well as several kinds of signal routing (wiring) elements can be implemented in this cellular space, and gave a construction method of a reversible counter machine out of these elements. An RE is a 4-input 4-output reversible element, and has two states. It is a kind of switching element that changes the path of an input signal depending on its state.

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 102–113, 2001.

In this paper, we show that any reversible Turing machine can be realized as a circuit composed only of REs in a systematic manner. In spite of the simplicity of an RE, the circuit obtained here is relatively concise and its operation is easy to be understood. Especially, there is no need to supply a clock signal to this circuit, and it contrasts sharply with a conventional logic circuit. We also discuss a simple reversible cellular automaton called P_3 proposed in [10], in which an RE can be embedded.

2 Preliminaries

2.1 A Reversible Sequential Machine

We first give a definition of a reversible sequential machine (RSM). As we shall see below, a reversible logic circuit composed of REs, as well as an RE itself, can be formulated as an RSM.

A *reversible sequential machine* (RSM) is a system defined by

$$M = (Q, \Sigma, \Gamma, q_1, \delta),$$

where Q is a finite non-empty set of states, Σ and Γ are finite non-empty sets of input and output symbols, respectively, and $q_1 \in Q$ is an initial state. $\delta : Q \times \Sigma \to Q \times \Gamma$ is a one-to-one mapping called a move function (hence $|\Sigma| \leq |\Gamma|$).

We can see that an RSM is "reversible" in the sense that, from the present state and the output of M, the previous state and the input are determined uniquely. A variation of an RSM $M = (Q, \Sigma, \Gamma, \delta)$, where no initial state is specified, is also called an RSM for convenience.

2.2 A Rotary Element and a Reversible Logic Circuit

A *rotary element* (RE) is a logic element depicted in Fig. 1. It has four input lines $\{n, e, s, w\}$ and four output lines $\{n', e', s', w'\}$, and has two states called H-state and V-state. All the values of inputs and outputs are either 0 or 1, i.e., $(n, e, s, w), (n', e', s', w') \in \{0, 1\}^4$. However, we restrict the input (and output) domain as $\{(0,0,0,0), (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)\}$, i.e., at most one "1" appears as an input (output) at a time. Hence, the operation of an RE is left undefined for the cases that signal 1's are given to two or more input lines. In order to explain its operation, we employ the following intuitive interpretation for it. Signals 1 and 0 are interpreted as existence and non-existence of a particle. An RE has a "rotating bar" to control the moving direction of a particle. When no particle exists, nothing happens on the RE. If a particle comes from a direction parallel to the rotating bar, then it goes out from the output line of the opposite side (i.e., it goes straight ahead) without affecting the direction of the bar (Fig. 2 (a)). On the other hand, if a particle comes from a direction orthogonal to the bar, then it makes a right turn, and rotates the bar by 90 degrees counterclockwise (Fig. 2 (b)).

We can define an RE as an RSM M_{RE}. Since the input $(0,0,0,0)$ has no effect on an RE, we omit it from the input alphabet for convenience. Further,

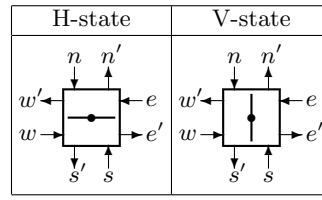


Fig. 1. Two states of a rotary element.

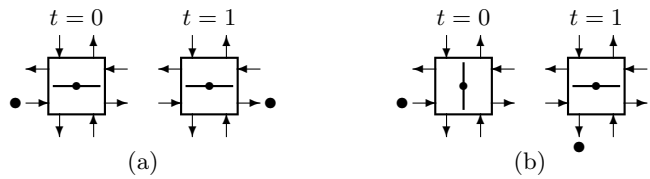


Fig. 2. Operations of a rotary element: (a) the parallel case (i.e., the coming direction of a particle is parallel to the rotating bar), and (b) the orthogonal case.

we denote the sets of input and output alphabets of M_{RE} as $\{n, e, s, w\}$ and $\{n', e', s', w'\}$ instead of $\{(1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)\}$.

M_{RE} is defined by

$$M_{\mathrm{RE}} = (\{\boxdash, \boxplus\}, \{n, e, s, w\}, \{n', e', s', w'\}, \delta_{\mathrm{RE}}),$$

where the move function δ_{RE} is shown in Table 1 (for instance, if the present state is $\boxdash$ and a particle comes from the input line n, then the state becomes $\boxplus$ and a particle goes out from w'). We can see that the operation of an RE is reversible. It has also a bit-conserving property, i.e., the number of 1's is conserved between inputs and outputs, since a particle is neither annihilated nor newly created.

Table 1. The move function δ_{RE} of a rotary element M_{RE}.

Present state	Input			
	n	e	s	w
H-state: $\boxdash$	$\boxplus\ w'$	$\boxdash\ w'$	$\boxplus\ e'$	$\boxdash\ e'$
V-state: $\boxplus$	$\boxplus\ s'$	$\boxdash\ n'$	$\boxplus\ n'$	$\boxdash\ s'$

An *RE-circuit* is a one composed only of REs satisfying the following condition: each output of an RE can be connected at most one input of some other (or may be the same) RE, i.e., "fan-out" of an output is not allowed. It is also easy to formulate each RE-circuit as an RSM.

2.3 Logical Universality of a Rotary Element

It has been shown that a Fredkin gate can be realized by an RE circuit as in in Fig.3 [9]. Since a Fredkin gate is logically universal [3], an RE is also universal. However, in the following, we do not use this construction method.

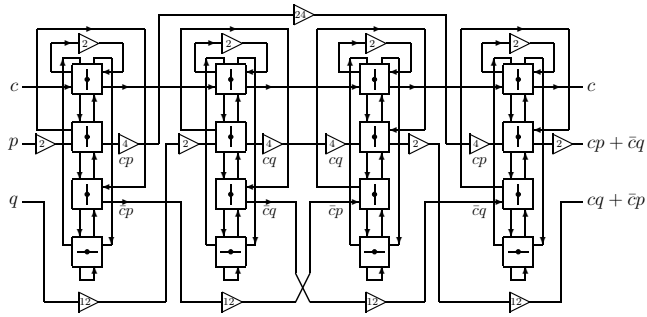


Fig. 3. Realization of a Fredkin gate as an RE circuit [9]. Small triangles are delay elements, where the number written inside of each triangle indicates its delay time. (Note that delay elements can also be implemented by using REs as in Fig.2(a).)

3 A Reversible Turing Machine Composed of REs

3.1 A Reversible Turing Machine

We first define a one-tape Turing machine and its reversible version. We use quadruple formalism [1] of a Turing machine in order to define a reversible one.

Definition 1. *A* one-tape Turing machine *(TM) is a system defined by*

$$T = (Q, S, q_0, q_f, s_0, \delta),$$

where Q is a non-empty finite set of states, S is a non-empty finite set of symbols, q_0 is an initial state ($q_0 \in Q$), q_f is a final state ($q_f \in Q$), s_0 is a special blank symbol ($s_0 \in S$), and δ is a move relation which is a subset of $(Q \times S \times S \times Q) \cup (Q \times \{/\} \times \{-, 0, +\} \times Q)$. Each element of δ is called a quadruple, and either of the form $[q_r, s, s', q_t] \in (Q \times S \times S \times Q)$ or $[q_r, /, d, q_t] \in (Q \times \{/\} \times \{-, 0, +\} \times Q)$. The symbols "$-$", "0", and "$+$" denote "left-shift", "zero-shift", and "right-shift", respectively. $[q_r, s, s', q_t]$ means that if T reads the symbol s in the state q_r, then write s' and go to the state q_t. $[q_r, /, d, q_t]$ means that if T is in the state q_r, then shift the head to the direction d and go to the state q_t.

Let $\alpha_1 = [p_1, b_1, c_1, p'_1]$ and $\alpha_2 = [p_2, b_2, c_2, p'_2]$ be two quadruples in δ. We say α_1 and α_2 overlap in domain *iff*

$$p_1 = p_2 \ \wedge \ [\ b_1 = b_2 \ \vee \ b_1 = / \ \vee \ b_2 = /\ ].$$

We say α_1 and α_2 overlap in range *iff*

$$p'_1 = p'_2 \ \wedge \ [\ c_1 = c_2 \ \vee \ b_1 = / \ \vee \ b_2 = /\ ].$$

A quadruple α is said to be deterministic *(in δ) iff there is no other quadruple in δ with which α overlaps in domain. On the other hand, α is said to be* reversible *(in δ) iff there is no other quadruple in δ with which α overlaps in range. T is called deterministic (reversible, respectively) iff every quadruple in δ is deterministic (reversible). (In what follows, we consider only deterministic Turing machines.)*

Theorem 1. [1] *For any one-tape Turing machine, there is a reversible three-tape Turing machine which simulates the former.*

Theorem 2. [6] *For any one-tape Turing machine, there is a reversible (semi-infinite) one-tape two-symbol Turing machine which simulates the former.*

Theorem 1 shows computation-universality of deterministic reversible three-tape Turing machines. Theorem 2 is useful for giving a simple construction method of a reversible TM out of REs.

3.2 Constructing a Tape Unit

In order to construct a reversible one-tape two-symbol Turing machine out of REs, we first deign a *tape cell module* (TC-module) as an RE-circuit. A TC-module simulates one tape square of a two-symbol reversible TM. It can store a symbol 0 or 1 written on the square and an information whether the tape head is on the square or not. It is formulated as an RSM M_{TC} defined below.

$$
\begin{aligned}
&M_{\mathrm{TC}} = (Q_{\mathrm{TC}}, \Sigma_{\mathrm{TC}}, \Gamma_{\mathrm{TC}}, \delta_{\mathrm{TC}})\\
&Q_{\mathrm{TC}} = \{(h,s) \mid h,s \in \{0,1\}\}\\
&\Sigma_{\mathrm{TC}} = \{R, Rc0, Rc1, W, Wc, SR, SRI, SRc, SL, SLI, SLc, E0, E1, Ec\}\\
&\Gamma_{\mathrm{TC}} = \{x' \mid x \in \Sigma_{\mathrm{TC}}\}
\end{aligned}
$$

δ_{TC} is defined as follows, where $s \in \{0,1\}$ and $y \in \Sigma_{\mathrm{TC}} - \{SRI, SLI\}$:

$$
\begin{aligned}
\delta_{\mathrm{TC}}((0,s),y) &= ((0,s),y') && (1)\\
\delta_{\mathrm{TC}}((0,s),SRI) &= ((1,s),SRc') && (2)\\
\delta_{\mathrm{TC}}((0,s),SLI) &= ((1,s),SLc') && (3)\\
\delta_{\mathrm{TC}}((1,0),R) &= ((1,0),Rc0') && (4)\\
\delta_{\mathrm{TC}}((1,1),R) &= ((1,1),Rc1') && (5)\\
\delta_{\mathrm{TC}}((1,0),W) &= ((1,1),Wc') && (6)\\
\delta_{\mathrm{TC}}((1,1),W) &= ((1,0),Wc') && (7)\\
\delta_{\mathrm{TC}}((1,s),SR) &= ((0,s),SRI') && (8)\\
\delta_{\mathrm{TC}}((1,s),SL) &= ((0,s),SLI') && (9)\\
\delta_{\mathrm{TC}}((1,0),E0) &= ((1,0),Ec') && (10)\\
\delta_{\mathrm{TC}}((1,1),E1) &= ((1,1),Ec') && (11)
\end{aligned}
$$

M_{TC} has the state set $\{(h,s) \mid h,s \in \{0,1\}\}$. The state (h,s) represents that the symbol s is written on the tape square, and that the tape head is on this cell (if $h = 1$) or not (if $h = 0$). There are 14 input and 14 output symbols. In the following construction of an RE-circuit, there are also 14 input and 14

Table 2. Fourteen input lines of an RSM M_{TC}.

Signal Line	Meaning of a Signal
R	Read the symbol at the head position.
$Rc0$	A read operation is completed with the result 0.
$Rc1$	A read operation is completed with the result 1.
W	Write a complementary symbol at the head position (i.e., if the current symbol is 1, then write 0, else write 1).
Wc	A write operation is completed.
SL	Shift the head position to the left.
SLI	Set the head position at the cell immediately to the left.
SLc	A left-shift operation is completed.
SR	Shift the head position to the right.
SRI	Set the head position at the cell immediately to the right.
SRc	A right-shift operation is completed.
$E0$	Reversibly erase the information 0 kept by the finite-state control by referring the symbol 0 at the head position.
$E1$	Reversibly erase the information 1 kept by the finite-state control by referring the symbol 1 at the head position.
Ec	A reversible erasure (i.e., merge) operation is completed.

output lines corresponding to these symbols. The roles of the 14 input lines are shown in Table 2. To each input line $x \in \Sigma_{\mathrm{TC}}$ there corresponds an output line $x' \in \Gamma_{\mathrm{TC}}$ in the one-to-one manner.

M_{TC} acts as follows according to the move function δ_{TC}.

(I) The case $h = 0$: (i) If a signal 1 arrives at the input line $y \in \Sigma_{\mathrm{TC}} - \{SRI, SLI\}$, then it simply goes out from y' without affecting the state of M_{TC} by (1). (ii) If a signal arrives at the line SRI (or SLI, respectively), then the head position is set to this tape cell, and a completion (i.e., response) signal for the shift-right (shift-left) operation goes out from the line SRc' (SLc') by (2) (or (3)).

(II) The case $h = 1$: (i) If a signal 1 arrives at the line R and if $s = 0$ (or $s = 1$, respectively), then a response signal goes out from $Rc0'$ ($Rc1$) by (4) (or (5)), performing a read operation. (ii) If a signal arrives at the line W and if $s = 0$ (or $s = 1$, respectively), then s is set to 1 (or 0) and a response signal goes out from W' by (6) (or (7)), performing an operation of writing a complementary symbol. (iii) If a signal arrives at the line SR (or SL, respectively), then h is set to 0 and a response signal goes out from SRI' (or SLI') by (8) (or (9)). (iv) If a signal arrives at the line $E0$ (or $E1$, respectively) and $s = 0$ ($s = 1$), then a response signal goes out from Ec' by (10) (or (11)), performing a "reversible erasure" of one bit of information by referring the symbol at the head position (usage of this operation is explained later).

Fig. 4 shows a TC-module, a realization of a tape cell M_{TC} as an RE-circuit. In order to explain the operations of a TC-module, it is convenient to consider an *RE-column* shown in Fig. 5 (a). It consists of $k+1$ REs, and has $2k$ input lines and $2k$ output lines ($k \in \{1, 2, \cdots\}$). We assume all REs except the bottom one (indicated by x) are initially set to V-states (the bottom RE may be either H- or

V-state). Further assume a signal 1 is given to at most one input line. Then, an RE-column acts like an RSM shown in Fig. 5 (b), where the input and output alphabets are $\{l_1, \cdots, l_k, r_1, \cdots, r_k\}$ and $\{l'_1, \cdots, l'_k, r'_1, \cdots, r'_k\}$, respectively, and the state set is $\{\boxed{\leftarrow}, \boxed{\uparrow}\}$ which matches that of the bottom RE. Though there are $k+1$ REs, we can consider the RE-column as if a two-state machine. Because all the REs except the bottom one are reset to V-states when a signal 1 goes out from some output line. For example, if it is in the state $\boxed{\leftarrow}$ and the input is l_j, then after *some* time steps the state becomes $\boxed{\uparrow}$ and gives an output l'_j. In what follows, we write $x = 1$ (or *marked*) if it is $\boxed{\leftarrow}$, and $x = 0$ (or *unmarked*) if it is $\boxed{\uparrow}$.

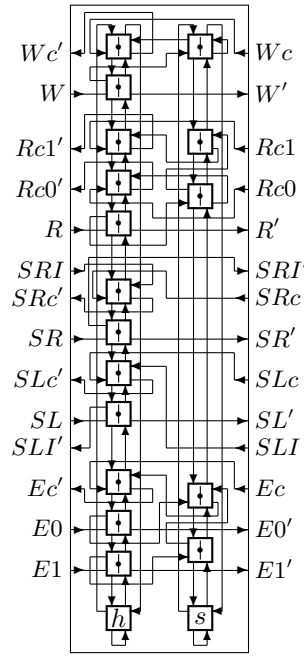
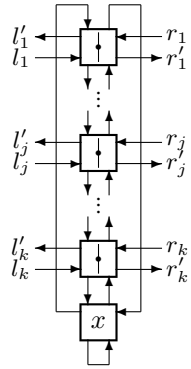


Fig. 4. A TC-module: a realization of a tape cell M_{TC} as an RE-circuit.

A TC-module consists of two RE-columns, i.e., left and right ones which correspond to h and s respectively. We can verify that the TC-module acts as M_{TC} by testing all the cases of inputs and states. For example, consider the case that a signal is given to R. If $h = 0$ then the signal eventually goes out from R' without affecting the states h and s. If $h = 1$, the signal first sets the state h to 0, and then enters the third RE of the right RE-column. There are two sub-cases: $s = 0$ and $s = 1$. If $s = 0$, the signal goes out from the right side of the third RE without affecting s, and enters the fourth RE of the left RE-column. It then sets h to 1, and finally goes out from $Rc0'$. If $s = 1$, the signal first goes out from the left side of the third RE setting s to 0, and enters the second RE of the right



(a)

State x	Input l_j	Input r_j
H-state: ⊡	⊡ l'_j	⊡ l'_j
V-state: ⊡	⊡ r'_j	⊡ r'_j

(b)

Fig. 5. (a) An RE-column, and (b) its move function ($j \in \{1, \cdots, k\}$).

RE-column. It restores both s and h to 1 and finally goes out from $Rc1'$. It is also easy to verify other cases.

An entire circuit for a tape unit can be obtained by placing infinite number of TC-modules in a row, and connecting input and output lines between adjacent TC-modules as shown in the right-half of Fig. 8. From the move function of the TC-module, we can see that by giving a signal 1 to one of the lines $R, W, SL, SR, E0$, or $E1$ of the leftmost TC-module in the semi-infinite array, it can correctly simulate each operation on the tape unit.

3.3 Constructing a Finite-State Control

We now design an RE-circuit that simulates a finite-state control of a given one-tape two-symbol reversible Turing machine T. This circuit is called a *finite-state control module* (FC-module). It is constructed in a similar manner given in [8].

Each quadruple of T performs either a read/write (i.e., R and/or W) operation, or a head-shift (SL or SR) operation to a tape unit. In addition, in order to realize a FC-module as an RE-circuit, we need a "reversible erasure" operation that erases an information kept by the FC-module. This is due to the following reason. When a read operation is performed, the information of the read symbol (0 or 1) is distinguished by some different states of the FC-module of T. If such an information is never erased, then the total amount of the information grows indefinitely, and thus an FC-module needs infinite number of states. Hence, such information should be reversibly erased (i.e., merged) by referring the read symbol itself each time a read operation is performed. This operation is done by giving a signal to the line $E0$ or $E1$ of the leftmost TC-module.

To explain a construction method of an FC-module, we consider a simple example of a one-tape two-symbol reversible Turing machine

$$T_{2n} = (\{q_1, \cdots, q_{16}\}, \{0, 1\}, q_1, q_{16}, 0, \delta_{2n})$$

having the following quadruples as δ_{2n}.

$[q_1, 0, 0, q_2]$
$[q_2, /, +, q_3]$
$[q_3, 0, 0, q_{16}]$
$[q_3, 1, 0, q_4]$
$[q_4, /, +, q_5]$
$[q_5, 0, 0, q_6]$
$[q_5, 1, 1, q_4]$
$[q_6, /, +, q_7]$
$[q_7, 0, 1, q_8]$
$[q_7, 1, 1, q_6]$
$[q_8, /, +, q_9]$
$[q_9, 0, 1, q_{10}]$
$[q_{10}, /, +, q_{11}]$
$[q_{11}, 0, 0, q_{12}]$
$[q_{12}, /, -, q_{13}]$
$[q_{13}, 0, 0, q_{14}]$
$[q_{13}, 1, 1, q_{12}]$
$[q_{14}, /, -, q_{15}]$
$[q_{15}, 0, 1, q_2]$
$[q_{15}, 1, 1, q_{14}]$

It is easy to verify that T_{2n} is reversible. It computes the function $f(n) = 2n$ for a unary input as shown in Fig. 6.

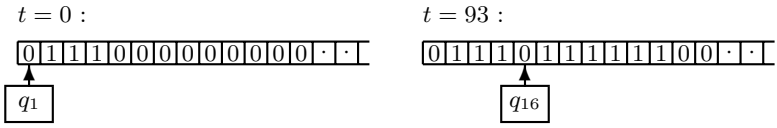


Fig. 6. Computing the function $f(n) = 2n$ by a reversible Turing machine T_{2n}.

An FC-module for T_{2n} is shown in Fig. 7. The quadruples of T_{2n} are executed and controlled by a matrix of REs. Namely, each column corresponds to each state of T_{2n}, and five rows correspond to five operations of write, read, shift-right, shift-left, and reversible erasure. At first, all the REs of the FC-module are in H-states.

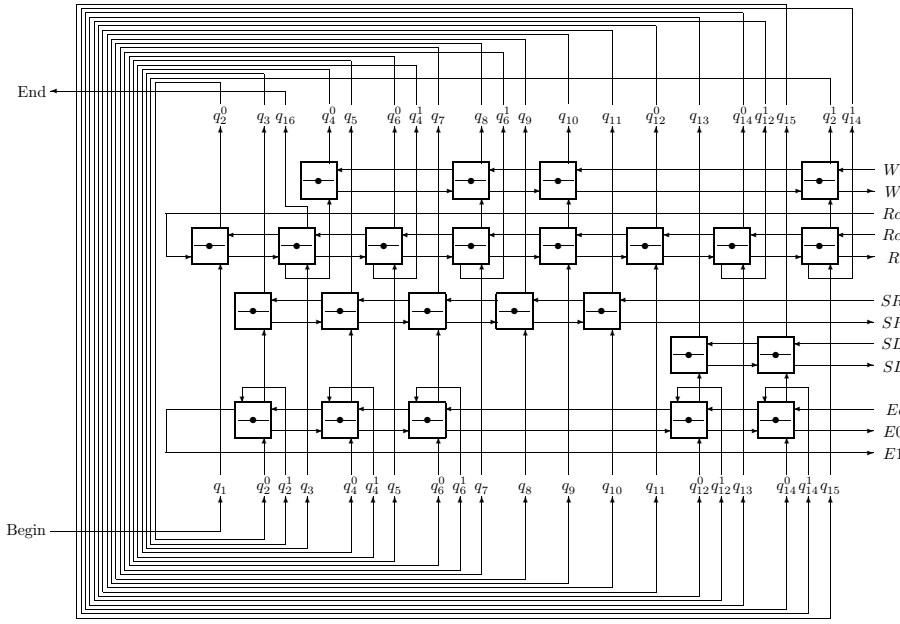


Fig. 7. The finite-state control module (FC-module) for T_{2n}.

The FC-module executes an operation as follows. First, consider the case of a shift-right operation. If the present state of T_{2n} is, e.g., q_8, then a signal 1 is put on the line q_8 in the lower part of Fig. 7. The signal goes upward to the RE on the row of a shift-right operation. After changing the RE to the V-state, the signal turns right and goes out from SR'. If a signal returns back from SRc, it restores the state of the RE to the H-state, and goes upward on the column.

Next, consider the case of a read operation. For example, if the present state is q_{13}, a signal 1 goes upward along the line up to the RE on the row of a read operation. Changing the RE to the V-state as in the previous case, it turns right and goes out from R'. A signal will return back from $Rc0$ or $Rc1$. It then restores the state of the RE, and goes upward taking a different path depending on the completion signal $Rc0$ or $Rc1$.

Generally, just after a read operation, a reversible erasure operation should be performed. It is in fact an inverse of a read operation. The bottom row of REs shown in Fig. 7 realizes this operation.

Appropriately connecting the vertical lines in the upper part of the figure to the ones in the lower part, state transition of T_{2n} is also realized. Further, the input line "Begin" is connected to the line of the initial state q_1, and the line of the final state q_{16} is to the output line "End". (From the method given in [6] we can assume that an initial state of a constructed reversible TM does not appear as the fourth element of a quadruple (hence it appears only at time 0)).

3.4 An RE-circuit Realizing an RTM

By connecting an FC-module with the tape module appropriately, we can obtain the whole circuit of a given reversible Turing machine. Fig. 8 shows the RE-circuit for T_{2n}. By giving a signal 1 to the "Begin" input it starts to compute.

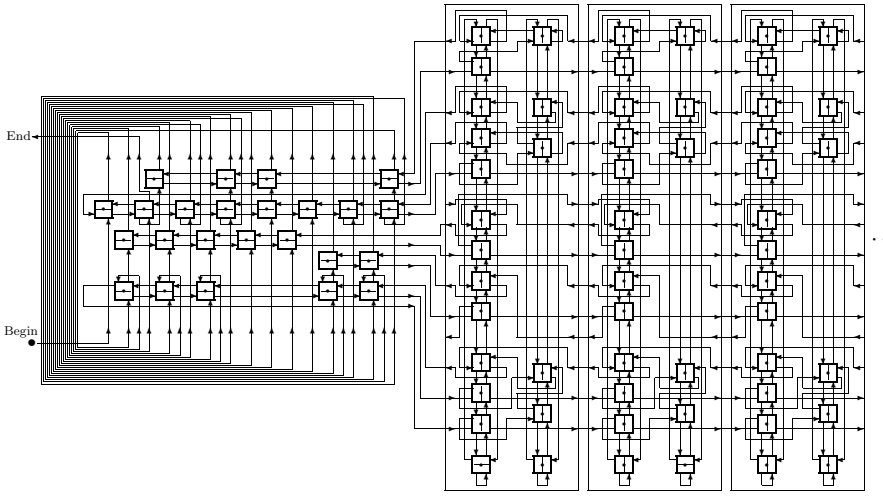


Fig. 8. The whole RE-circuit realizing T_{2n}.

4 A Simple Universal Reversible Cellular Automaton

In [10], a 3^4-state reversible partitioned cellular automaton (RPCA) P_3 was proposed, and it was shown that any reversible counter machine can be embedded in the P_3 space. This improves the previous result in [8] on the number of states. Each cell of P_3 has four parts, and each part has a state set $\{0, 1, 2\}^4$ (0, 1 and 2 are indicated by a blank, ○ and ●). Its local transition function is shown in Fig. 9. Reversibility of P_3 is verified by checking there is no pair of distinct rules having the same right-hand side. Fig. 10 shows an RE embedded in P_3 space, where a single ● acts as a signal. Hence, the function of an RE can be decomposed into much simpler reversible local rules of P_3.

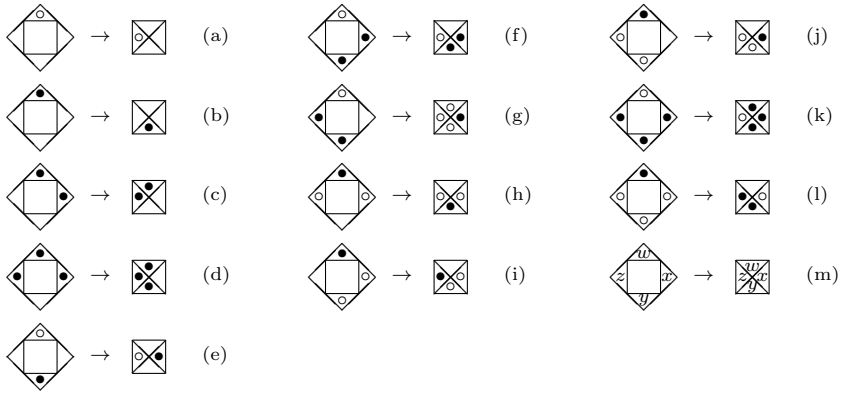


Fig. 9. The set of 13 rule schemes (representing 81 rules) of the rotation-symmetric 3^4-state RPCA P_3. Each rule scheme of (a)–(l) stands for four rules obtained by rotating the left- and right-hand sides of it by 0, 90, 180 and 270 degrees. The rule scheme (m) represents 33 rules not specified by (a)–(l) ($w, x, y, z \in$ {blank, ○, ●}).

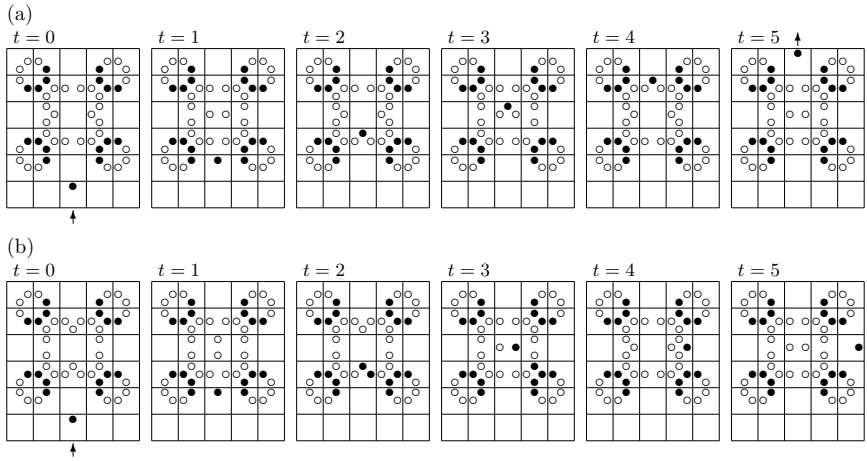


Fig. 10. An RE in the P_3 space: (a) the parallel case, and (b) the orthogonal case.

5 Concluding Remarks

In this paper, we showed that any reversible Turing machine can be realized as a circuit composed only of rotary elements. If we use logic gates such as AND, OR, Fredkin gate, or Toffoli gate, some synchronization mechanism should be provided to make two or more incoming signals interact properly. In a conventional circuit, it is solved by giving a clock signal. However, in the case of a rotary element, there is only one incoming signal to each element, which interacts with rotating bar of the element. Therefore it acts in a somewhat asynchronous manner, and thus there is no need to provide a clock signal. Hence, construction of a whole circuit becomes relatively simple.

There have been known several simple universal models of reversible cellular automata (e.g., [5,7,8,10]). For example, a Fredkin gate can be embedded in the cellular spaces in the references [5] and [7]. Though these models have very small number of states, they need an infinite configuaraion to realize a universal computer. On the other hand, in the models proposed in [8,10], a reversible counter machine can be realized as a finite configuration by using rotary elements and some other elements implemented in these cellular spaces. It is left for the future study whether there are still other methods to implement a universal computer in a simple reversible cellular space, and whether further simplification of a rotary element is possible.

References

1. Bennett, C.H., Logical reversibility of computation, *IBM J. Res. Dev.*, **17**, 525–532 (1973).
2. Feynman, R.P., *Feynman Lectures on Computation* (eds., A.J.G. Hey and R.W. Allen), Perseus Books, Reading, Massachusetts (1996).
3. Fredkin, E. and Toffoli, T., Conservative logic, *Int. J. Theoret. Phys.*, **21**, 219–253 (1982).
4. Gruska, J., *Quantum Computing*, McGraw-Hill, London (1999).
5. Margolus, N., Physics-like model of computation, *Physica*, **10D**, 81–95 (1984).
6. Morita, K., Shirasaki, A. and Gono, Y., A 1-tape 2-symbol reversible Turing machine, *Trans. IEICE Japan*, **E-72**, 223–228 (1989).
7. Morita, K., and Ueno, S., Computation-universal models of two-dimensional 16-state reversible cellular automata, *IEICE Trans. Inf. & Syst.*, **E75-D**, 141–147 (1992).
8. Morita, K., Tojima, Y. and Imai, K., A simple computer embedded in a reversible and number-conserving two-dimensional cellular space, *Multiple-Valued Logic*, (in press). Movie: `http://www.ke.sys.hiroshima-u.ac.jp/~morita/p4/`
9. Morita, K., A new universal logic element for reversible computing, *Technical Report of IEICE Japan*, COMP99-94 (2000).
10. Morita, K. and Ogiro T., Embedding a counter machine in a simple reversible 2-D cellular space, *Proc. Int. Workshop on Cellular Automata*, Osaka, 30–31 (2000). Movie: `http://www.ke.sys.hiroshima-u.ac.jp/~morita/p3/`
11. Păun, Gh., Rozenberg, G. and Salomaa, A., *DNA Computing*, Springer-Verlag, Berlin (1998).
12. Toffoli, T., Reversible computing, in *Automata, Languages and Programming*, Springer-Verlag, LNCS-85, 632–644 (1980).

Some Applications of the Decidability of DPDA's Equivalence

Géraud Sénizergues

LaBRI, Université de Bordeaux I
ges@labri.u-bordeaux.fr
http://www.dept-info.labri.u-bordeaux.fr/~ges/

Abstract. The *equivalence* problem for deterministic pushdown automata has been shown decidable in [Sén97c,Sén97a,Sén97b,Sén01,Sti99]. We give some applications of this decidability result to other problems arising in the following areas of theoretical computer science:
- programming languages theory
- infinite graph theory
- Thue-systems

1 Introduction

We describe, in this extended abstract, several applications of the decidability of the equivalence problem for dpda (we denote this problem by Eq(D,D) in the sequel), in the following areas

- programming languages theory
- infinite graph theory
- Thue-systems

We have left out of this description applications to other equivalence problems in formal language theory; such applications are already described in [Sén01, p.155-156].
What we call an *application* may be either a true application of theorem 1 or a result obtained by a method which is a variant of the method used in [Sén01]. In many cases the "application" to a problem P will follow from a reduction from problem P to Eq(D,D) which already existed in the literature (i.e. the reduction was established before Eq(D,D) was solved). In some cases (corollary 5, theorem 14, theorem 16) we shall give a *new* reduction of some problem P to Eq(D,D). In all cases we try to sketch the landscape around every application.

2 Deterministic Pushdown Automata

In this section we just recall the basic definitions on dpda and the result about decidability of the equivalence problem. More details about either the proof of this result or the historical development of the works around this problem can be found elsewhere (see [Sén97a] for the solution, [Sén01, section 1.1] for a general historical outline and [Sén00a] for an historical perspective on the proof-systems used in the solution).

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 114–132, 2001.

2.1 Pushdown Automata

A *pushdown automaton* on the alphabet X is a 7-tuple $\mathcal{M}=<X, Z, Q, \delta, q_0, z_0, F>$ where Z is the finite stack-alphabet, Q is the finite set of states, $q_0 \in Q$ is the initial state, z_0 is the initial stack-symbol, F is a finite subset of QZ^*, the set of *final* configurations, and δ, the transition function, is a mapping $\delta : QZ \times (X \cup \{\epsilon\}) \to \mathcal{P}_f(QZ^*)$.

Let $q, q' \in Q, \omega, \omega' \in Z^*, z \in Z, f \in X^*$ and $a \in X \cup \{\epsilon\}$; we note $(qz\omega, af) \longmapsto_{\mathcal{M}} (q'\omega'\omega, f)$ if $q'\omega' \in \delta(qz, a)$. $\overset{*}{\longmapsto}_{\mathcal{M}}$ is the reflexive and transitive closure of $\longmapsto_{\mathcal{M}}$.

For every $q\omega, q'\omega' \in QZ^*$ and $f \in X^*$, we note $q\omega \overset{f}{\longrightarrow}_{\mathcal{M}} q'\omega'$ iff $(q\omega, f) \overset{*}{\longmapsto}_{\mathcal{M}} (q'\omega', \epsilon)$.

$\mathcal{M}$ is said *deterministic* iff, for every $z \in Z, q \in Q, x \in X$:

$$\mathrm{Card}(\delta(qz, \epsilon)) \in \{0, 1\} \tag{1}$$

$$\mathrm{Card}(\delta(qz, \epsilon)) = 1 \Rightarrow \mathrm{Card}(\delta(qz, x)) = 0, \tag{2}$$

$$\mathrm{Card}(\delta(qz, \epsilon)) = 0 \Rightarrow \mathrm{Card}(\delta(qz, x)) \leq 1. \tag{3}$$

A configuration $q\omega$ of $\mathcal{M}$ is said ϵ-bound iff there exists a configuration $q'\omega'$ such that $(q\omega, \epsilon) \longmapsto_{\mathcal{M}} (q'\omega', \epsilon)$; $q\omega$ is said ϵ-free iff it is not ϵ-bound.
A pda $\mathcal{M}$ is said *normalized* iff, it fulfills conditions (1), (2) (see above) and (4),(5),(6):

$$q_0 z_0 \text{ is } \epsilon - \text{free} \tag{4}$$

and for every $q \in Q, z \in Z, x \in X$:

$$q'\omega' \in \delta(qz, x) \Rightarrow \mid \omega' \mid \leq 2, \tag{5}$$

$$q'\omega' \in \delta(qz, \epsilon) \Rightarrow \mid \omega' \mid = 0. \tag{6}$$

The *language recognized* by $\mathcal{M}$ is

$$\mathrm{L}(\mathcal{M}) = \{w \in X^* \mid \exists c \in F, q_0 z_0 \overset{w}{\longrightarrow}_{\mathcal{M}} c\}.$$

We call *mode* every element of $QZ \cup \{\epsilon\}$. For every $q \in Q, z \in Z$, qz is said *ϵ-bound* (respectively ϵ-free) iff condition (2) (resp. condition (3)) in the above definition of deterministic automata is realized. The mode ϵ is said ϵ-free. We define a mapping $\mu : QZ^* \to QZ \cup \{\epsilon\}$ by

$$\mu(\epsilon) = \epsilon \text{ and } \mu(qz \cdot \omega) = qz,$$

for every $q \in Q, z \in Z, \omega \in Z^*$. For every $c \in QZ^* \cup \{\epsilon\}$, $\mu(c)$ is called the mode of c. The configuration c is said ϵ-bound (resp. ϵ-free) if and only if $\mu(c)$ has the corresponding property.

2.2 The Equivalence Problem

The *equivalence problem* for dpda was raised in [GG66]. It is the following decision problem:

INSTANCE: Two dpda $\mathcal{A}, \mathcal{B}$, over the same terminal alphabet X.
QUESTION: $\mathrm{L}(\mathcal{A}) = \mathrm{L}(\mathcal{B})$?

Theorem 1. *The* equivalence problem *for dpda is decidable.*

The result is exposed in [Sén97c,Sén97a] and proved in [Sén97b,Sén01], see in [Sti99] some simplifications.

3 Programming Languages

3.1 Semantics

The equivalence problem for program schemes
Let us say that two programs P, Q are *equivalent* iff, on every given input, either they both diverge or they both converge and compute the same result. It would be highly desirable to find an algorithm deciding this equivalence between programs since, if we consider that P is really a program and Q is a specification, this algorithm would be a "universal program-prover". Unfortunately one can easily see that, as soon as the programs P, Q compute on a sufficiently rich structure (for example the ring of integers), this notion of equivalence is undecidable. Nevertheless, this seemingly hopeless dream lead many authors to analyze the *reason why* this problem is undecidable and the suitable *restrictions* (either on the shape of programs or on the meaning of the basic operations they can perform) which might make this equivalence decidable. Informally, one can define an *interpretation* as an "universe of objects together with a certain definite meaning for each program primitive as a function on this universe" and a *program scheme* as a "program without interpretation"([Mil70, p.205,lines 5-13]). Several precise mathematical notions of "interpretation" and "program schemes" were given and studied ([Ian60],[Rut64],[Pat67],[Kap69],[Mil70],[LPP70],[GL73],[Niv75],[Ros75],[Fri77], [Cou78a],[Cou78b],[Gue81], see [Cou90b] for a survey). Many methods for either transforming programs or for proving properties of programs were established but, concerning the equivalence problem, the results turned out to be mostly negative: for example, in [LPP70, p.221, lines 24-26], the authors report that "for almost any reasonable notion of equivalence between computer programs, the two questions of equivalence and nonequivalence of pairs of schemas are *not* partially decidable". Nevertheless, two kinds of program schemes survived all these studies:

- the *monadic recursion schemes* where a special ternary function `if-then-else` has the fixed usual interpretation: in [GL73] the equivalence-problem for such schemes is reduced to the equivalence problem for dpda and in [Fri77] a reduction in the opposite direction is constructed,

– the *recursive polyadic program schemes*: in [Cou78a,Cou78b], following a representation principle introduced in [Ros75], the equivalence-problem for such schemes is reduced to the equivalence problem for dpda and conversely.

Corollary 2. *The equivalence problem for monadic recursion schemes (with interpreted* `if-then-else`*), is decidable.*

This follows from theorem 1 and the reduction given in [GL73].

Corollary 3. *The equivalence problem for recursive polyadic program schemes (with completely uninterpreted function symbols) is decidable.*

This follows from theorem 1 and the reduction given in [Cou78a, theorem 3.25] or the reduction given in [Gal81, corollary 4.4].

Interpretation by trees Let us precise now what a "recursive polyadic" program scheme is. Let (F, ρ) be some *ranked alphabet* i.e. a set F and a map $\rho : F \to \mathbb{N}$. A *tree* over the ranked alphabet (F, ρ) is a partial mapping $t : (\mathbb{N} - \{0\})^* \to F$ satisfying the following conditions:
C1: $\mathrm{dom}(t) \subseteq (\mathbb{N} - \{0\})^*$ is prefix-closed, i.e. if $\alpha \cdot \beta \in \mathrm{dom}(t)$, then $\alpha \in \mathrm{dom}(t)$.
C2: if $\rho(t(\alpha)) = k$, then, for every $i \in \mathbb{N}$, $\alpha i \in \mathrm{dom}(t) \Leftrightarrow 1 \leq i \leq k$.

We denote by $\mathrm{M}(F)$ (resp. $\mathrm{M}^\infty(F)$) the set of finite trees (resp. the set of all trees, finite or infinite) over F. With every symbol f of arity $k = \rho(f)$ is associated a k-ary operation $\hat{f} : \mathrm{M}(F)^k \to \mathrm{M}(F)$ defined by $\hat{f}(t_1, \ldots, t_k)$ is the unique tree which has a root labelled by f and which has k subtrees at depth 1, the ith subtree being t_i.

A *system of algebraic equations* Σ over F is defined as follows:
let $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ be a ranked alphabet of "unknowns" (we denote by k_i the arity of φ_i) and let $X = \{x_1, \ldots, x_m\}$ be an alphabet of "variables", which are 0-ary symbols. Σ is a set of n equations of the form

$$\varphi_i(x_1, x_2, \ldots, x_{k_i}) = T_i \tag{7}$$

where every T_i is some element of $\mathrm{M}(F \cup \Phi, X)$. A *solution* of Σ is a tuple $(t_1, \ldots, t_n) \in (\mathrm{M}^\infty(F \cup \Phi, X))^n$ such that, for every $1 \leq i \leq n$:

$$\hat{t_i}(x_1, x_2, \ldots, x_m) = \hat{T_i}(x_1, x_2, \ldots, x_m) \tag{8}$$

where, $\hat{t_i}$ is the operation obtained by interpreting every symbol f by the corresponding operation $\hat{f}$ and $\hat{T_i}$ is the operation obtained by interpreting every symbol φ_i (resp. f) by the corresponding operation $\hat{t_i}$ (resp. $\hat{f}$). Σ will be said *in normal form* iff every tree T_i has its root labelled by an element of F. In such a case Σ has a *unique* solution. A program scheme is a pair (Σ, T) where Σ is an algebraic system of equations over F and T is a particular finite tree $T \in \mathrm{M}(F \cup \Phi, X)$. The *tree computed* by the scheme is then

$$t = \hat{T}(x_1, x_2, \ldots, x_m).$$

What corollary 3 precisely means is that the following problem is decidable:

INSTANCE: Two program schemes $(\Sigma_1, T_1), (\Sigma_2, T_2)$ (assumed in normal form)
QUESTION: Do these two schemes compute the same algebraic tree ?

Let us describe now some other F-magmas which extend $(\mathrm{M}^\infty(F), (\hat{f})_{f\in F})$. It turns out that corollary 3 is true, as well, in these structures.

Interpretation by formal power series
The idea of this link with formal power series is due to [Mat95]. Let F_2 be the field with 2 elements (i.e. $(\mathbb{Z}/2\mathbb{Z}, +, \cdot)$). Let $D = F_2[[X_1, X_2, \ldots, X_m, Y]]$ be the set of formal power series with $m+1$ commutative undeterminates and coefficients in F_2. We consider the two binary operations $\bar{f}, \bar{g}$ over D:

$$\bar{f}(S,T) = Y \cdot S^2 + Y^2 \cdot T^2; \;\; \bar{g}(S,T) = 1 + Y \cdot S^2 + Y^2 \cdot T^2.$$

Given an element $S \in D$ let us define the associated operation $\bar{S} : D^m \to D$ by

$$\bar{S}(S_1, S_2, \ldots, S_m) = S \circ (S_1, S_2, \ldots, S_m, Y)$$

i.e. the series obtained by substituting S_i to the undeterminate X_i and leaving Y unchanged.
Let Σ be a system of equations (7) over the ranked alphabet $F = \{f, g\}$, with $\rho(f) = \rho(g) = 2$. A solution in D of (7) is a n-tuple $(S_1, S_2, \ldots, S_n)$ such that for every $1 \leq i \leq n$:

$$\bar{S}_i(X_1, X_2, \ldots, X_m) = \bar{T}_i(X_1, X_2, \ldots, X_m) \tag{9}$$

where, $\bar{S}_i$ is the operation defined above and $\bar{T}_i$ is the operation obtained by interpreting every symbol φ_i (resp. f, g) by the corresponding operation $\bar{S}_i$ (resp by $\bar{f}, \bar{g}$).

Let us define a map $\mathcal{S} : \mathrm{M}^\infty(F, \{x_1, x_2, \ldots, x_m\}) \to F_2[[X_1, X_2, \ldots, X_m, Y]]$ by:for every $t_1, t_2 \in \mathrm{M}^\infty(F, \{x_1, x_2, \ldots, x_m\})$

$$\mathcal{S}(x_i) = X_i; \;\; \mathcal{S}(f(t_1,t_2)) = \bar{f}(\mathcal{S}(t_1), \mathcal{S}(t_2)); \;\; \mathcal{S}(g(t_1,t_2)) = \bar{g}(\mathcal{S}(t_1), \mathcal{S}(t_2)).$$

One can check that $\mathcal{S}$ is an injective homomorphism from the magma $(\mathrm{M}^\infty(F, \{x_1, x_2, \ldots, x_m\}), \hat{f}, \hat{g})$ into the magma $(F_2[[X_1, X_2, \ldots, X_m, Y]], \bar{f}, \bar{g})$. Moreover the map $\mathcal{S}$ is *compatible with substitution*, in the following sense: $\forall t \in \mathrm{M}^\infty(F, \{x_1, x_2, \ldots, x_m\}, \forall (u_1, \ldots, u_m) \in (\mathrm{M}^\infty(F, \{x_1, x_2, \ldots, x_m\})^m$,

$$\mathcal{S}(\hat{t}(u_1, \ldots, u_m)) = \overline{\mathcal{S}(t)}(\mathcal{S}(u_1), \ldots, \mathcal{S}(u_m)).$$

(This last property follows from the fact that $\bar{f}, \bar{g}$ were chosen to be *polynomial* operators).
The fact that $\mathcal{S}$ is an homomorphism and that it is compatible with substitution show that $\mathcal{S}$ maps every solution of Σ in $\mathrm{M}^\infty(F, \{x_1, x_2, \ldots, x_m\})$ to a solution of Σ in $F_2[[X_1, X_2, \ldots, X_m, Y]]$. The hypothesis that Σ is normal implies that it has a unique solution in $F_2[[X_1, X_2, \ldots, X_m, Y]]$ too. In conclusion, two algebraic systems of equations Σ_1, Σ_2 have the same solution (resp. the same first component of solution) in $F_2[[X_1, X_2, \ldots, X_m, Y]]$ iff they have the same solution (resp. the same first component of solution) in $\mathrm{M}^\infty(F, X)$.

Corollary 4. *The equality problem for series in* $F_2[[X_1, X_2, \ldots, X_m, Y]]$ *defined by an algebraic system of equations (with polynomial operators* $\bar{f}, \bar{g}$ *and with substitution) is* decidable.

Let us notice that for any finite field F_q (with $q = p^k$, p prime), one can introduce the polynomial operators:

$$\bar{f}(S_1, S_2, \ldots, S_p) = f + YS_1^p + \ldots + Y^j S_j^p + \ldots + Y^p S_p^p \quad (\text{ for } f \in F_q).$$

By the same type of arguments one obtains

Corollary 5. *The equality problem for series in* $F_q[[X_1, X_2, \ldots, X_m, Y]]$ *defined by an algebraic system of equations (with polynomial operators* $(\bar{f})_{f \in F_q}$ *and with substitution) is* decidable.

3.2 Types

Some works on *recursively defined parametric types* have raised the question whether it is possible to decide if two such types are equivalent or not. It has been shown in [Sol78] that this problem is reducible to the equivalence problem for dpda. Therefore we can state

Corollary 6. *The equivalence problem for recursively defined types is decidable.*

This follows from theorem 1 and the reduction given in [Sol78].

4 Graphs

We describe here several problems arising in the study of infinite graphs. Such infinite graphs arise naturally in computer science either as representing all the possible computations of a program (the infinite expansion of a recursive program scheme), or all the possible behaviours of a process, etc... They also arise in computational group theory as a geometrical view of the group (the so-called *Cayley*-graph of the group).
When the chosen class of graphs is well-related with pushdown automata, one can draw from either theorem 1 or its proof method some clue for solving decision problems on these graphs.

4.1 Bisimulations-Homomorphisms

Let X be a finite alphabet. We call *graph over* X any pair $\Gamma = (V_\Gamma, E_\Gamma)$ where V_Γ is a set and E_Γ is a subset of $V_\Gamma \times X \times V_\Gamma$. For every integer $n \in \mathbb{N}$, we call an n-graph every $(n+2)$-tuple $\Gamma = (V_\Gamma, E_\Gamma, v_1, \ldots, v_n)$ where (V_Γ, E_Γ) is a graph and $(v_1, \ldots, v_n)$ is a sequence of distinguished vertices: they are called the *sources* of Γ.
A 1-graph (V, E, v_1) is said to be *rooted* iff v_1 is a root of (V, E). Let Γ, Γ' be two n-graphs over X.

Definition 7 *$\mathcal{R}$ is a* simulation *from Γ to Γ' iff*

1. $\mathrm{dom}(\mathcal{R}) = V_\Gamma$,
2. $\forall i \in [1,n], (v_i, v'_i) \in \mathcal{R}$,
3. $\forall v, w \in V_\Gamma, v' \in V_{\Gamma'}, x \in X$, *such that* $(v,x,w) \in E_\Gamma$ *and* $v \mathcal{R} v'$,

$$\exists w' \in V_{\Gamma'} \text{ such that } (v', x, w') \in E_{\Gamma'} \text{ and } w \mathcal{R} w'.$$

$\mathcal{R}$ is a bisimulation *iff both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations.*

In the case where $n = 0$ this definition coincides with the classical one from [Par81], [Mil89].

Definition 8 *We call* homorphism *from Γ to Γ' any mapping: $h : V_\Gamma \to V_{\Gamma'}$ such that h is a bisimulation in the sense of definition 7.*
h is called an isomorphism *iff h is a bijective bisimulation.*

4.2 Some Classes of Infinite Graphs

Equational Graphs. The reader can find in [Cou89,Bau92] formal definitions of what an *equational* graph is. However, theorem 9 below gives a strong link with pda.
We call *transition-graph* of a pda $\mathcal{M}$, denoted $\mathcal{T}(\mathcal{M})$, the 0-graph:
$\mathcal{T}(\mathcal{M}) = (V_{\mathcal{T}(\mathcal{M})}, E_{\mathcal{T}(\mathcal{M})})$ where $V_{\mathcal{T}(\mathcal{M})} = \{q\omega \mid q \in Q, \omega \in Z^*, q\omega \text{ is } \epsilon - \text{free}\}$ and

$$E_{\mathcal{T}(\mathcal{A})} = \{(c, x, c') \in V_{\mathcal{T}(\mathcal{M})} \times V_{\mathcal{T}(\mathcal{M})} \mid c \xrightarrow{x}_{\mathcal{M}} c'\}. \tag{10}$$

We call *computation 1-graph* of the pda $\mathcal{M}$, denoted $(\mathcal{C}(\mathcal{M}), v_{\mathcal{M}})$, the subgraph of $\mathcal{T}(\mathcal{M})$ induced by the set of vertices which are accessible from the vertex $q_0 z_0$, together with the source $v_{\mathcal{M}} = q_0 z_0$.

Theorem 9. *Let $\Gamma = (\Gamma_0, v_0)$ be a rooted 1-graph over X. The following conditions are equivalent:*

1. *Γ is equational and has finite out-degree.*
2. *Γ is isomorphic to the computation 1-graph $(\mathcal{C}(\mathcal{M}), v_{\mathcal{M}})$ of some normalized pushdown automaton $\mathcal{M}$.*

A proof of this theorem is sketched in [Sén00b, Annex,p.94-96].

Algebraic Trees. Let (F, ρ) be some *ranked alphabet* i.e. a set F and a map $\rho : F \to \mathbb{N}$. The notion of a *tree* over the ranked alphabet (F, ρ) has been recalled in §3.1. With such a tree one associates a new alphabet

$$Y = \{[f,k] \mid f \in F, 1 \leq k \leq \rho(f)\}. \tag{11}$$

Let $\alpha \in \mathrm{dom}(t) : \alpha = i_1 i_2 \dots i_n$. The word associated with α is : $brch(\alpha) = [f_0, i_1][f_1, i_2] \dots [f_{n-1}, i_n]$ where α_j is the prefix of α of length j and $f_j = t(\alpha_j)$. The *branch-language* associated to t is then:

$$Brch(t) = \{brch(\alpha) \mid \alpha \in \mathrm{dom}(t)\}.$$

One can easily check that, if t, t' are trees over F, with no leaf, then t, t' are isomorphic iff $Brch(t) = Brch(t')$. The notion of *algebraic* tree over F can be defined in terms of algebraic equations in the magma of (infinite) trees over F: $t \in \mathrm{M}(F)$ is algebraic iff there exists some normal system of n equations Σ over F (see equation (7) in §3.1)such that, t is the first component of its unique solution. Anyway, the following characterization is sufficient for our purpose

Theorem 10. *Let t be a tree over the ranked alphabet (F, ρ), without any leaf. t is algebraic iff the language $Brch(t)$ is deterministic context-free.*

A more general version valid even for trees having some leaves is given in [Cou83, Theorem 5.5.1, point 2]. The trees considered above can be named *ordered* trees because the sons of every vertex α are given with a fixed order: $\alpha 1, \alpha 2, \ldots, \alpha k$. We call *unordered* tree the oriented graph $un(t)$, with vertices labelled over F and unlabelled edges, obtained by forgetting this ordering of the sons in an ordered tree t. An *algebraic unordered* tree is then a tree of the form $un(t)$ for some algebraic ordered tree t.

Automatic Graphs. The general idea underlying the notion of an *automatic* graph is that of a graph whose set of vertices is fully described sy some *rational* set of words and whose set of edges is fully described by some *rational* set of pairs of words. Several precise technical definitions following this general scheme have been studied in [Sén92,KN95,Pel97,CS99,BG00,Mor00,Ly00b,Ly00a]. This idea appeared first in the context of group theory in [ECH$^+$92].

Definition 1. *A deterministic 2-tape finite automaton (abbreviated 2-d.f.a. in the sequel) is a 5-tuple:*

$$\mathcal{M} =< X, Q, \delta, q_0, F >$$

where

- *X is a finite alphabet, the input-alphabet*
- *Q is a finite set, the set of states*
- *q_0 is a distinguished state , the initial state*
- *$F \subseteq Q$ is a set of distinguished states , the final states*
- *δ, the transition function, is a partial map from $Q \times (X \cup \{\#, \epsilon\})^2$ to Q (where $\#$ is a new letter not in X) fulfilling the restrictions:*
  1. *$\forall q \in Q, \delta(q, \epsilon, \epsilon)$ is undefined,*
  2. *$\forall q \in Q, \forall \boldsymbol{u} \in (X \cup \{\#\})^2, \forall a \in X \cup \{\#\}$, if $\delta(q, \epsilon, a)$ is defined, then*

$$\delta(q, \boldsymbol{u}) \text{ is defined } \Longrightarrow \boldsymbol{u} = (\epsilon, b), \text{ for some } b \in X \cup \{\#\}$$

  3. *$\forall q \in Q, \forall \boldsymbol{u} \in (X \cup \{\#\})^2, \forall a \in X \cup \{\#\}$, if $\delta(q, a, \epsilon)$ is defined, then*

$$\delta(q, \boldsymbol{u}) \text{ is defined } \Longrightarrow \boldsymbol{u} = (b, \epsilon), \text{ for some } b \in X \cup \{\#\}.$$

The notation $q \xrightarrow[\mathcal{M}]{(u_1,u_2)} q'$ means that there is some computation of the automaton $\mathcal{M}$ starting fron state q , reading the input $(u_1, u_2) \in (X \cup \{\#\})^* \times (X \cup \{\#\})^*$ and ending in state q'.

The language recognized by $\mathcal{M}$ is:

$$\mathrm{L}(\mathcal{M}) = \{(u_1, u_2) \in X^* \times X^*, \mid \exists q \in F, q_0 \xrightarrow[\mathcal{M}]{(u_1\#,u_2\#)} q\}.$$

We are interested in some restrictions on 2-d.f.a.
Let us consider the four situations:
$\exists q \in Q, q' \in F, p_1, p_2, u_1 \neq \epsilon, s_1, s_2 \in X^*$ such that

$$q_0 \xrightarrow[\mathcal{M}]{(p_1,p_2)} q \xrightarrow[\mathcal{M}]{(u_1,\epsilon)} q \xrightarrow[\mathcal{M}]{(s_1\#,s_2\#)} q' \tag{12}$$

$\exists q \in Q, q' \in F, p_1, p_2, u_2 \neq \epsilon, s_1, s_2 \in X^*$ such that

$$q_0 \xrightarrow[\mathcal{M}]{(p_1,p_2)} q \xrightarrow[\mathcal{M}]{(\epsilon,u_2)} q \xrightarrow[\mathcal{M}]{(s_1\#,s_2\#)} q' \tag{13}$$

$\exists q_1, q_1' \in Q, q' \in F, w_1, w_2, p_1, u_1 \neq \epsilon, s_1 \in X^*$ such that

$$q_0 \xrightarrow[\mathcal{M}]{(w_1,w_2\#)} q_1 \xrightarrow[\mathcal{M}]{(p_1,\epsilon)} q_1' \xrightarrow[\mathcal{M}]{(u_1,\epsilon)} q_1' \xrightarrow[\mathcal{M}]{(s_1\#,\epsilon)} q' \tag{14}$$

$\exists q_2, q_2' \in Q, q' \in F, w_1, w_2, p_2, u_2 \neq \epsilon, s_2 \in X^*$ such that

$$q_0 \xrightarrow[\mathcal{M}]{(w_1\#,w_2)} q_2 \xrightarrow[\mathcal{M}]{(\epsilon,p_2)} q_2' \xrightarrow[\mathcal{M}]{(\epsilon,u_2)} q_2' \xrightarrow[\mathcal{M}]{(\epsilon,s_2\#)} q' \tag{15}$$

The 2-d.f.a. $\mathcal{M}$ will be said *strictly balanced* iff neither of situations (12,13,14,15) is possible.
The 2-d.f.a. $\mathcal{M}$ will be said *balanced* iff neither of situations (12,13,15) is possible.

Given a deterministic graph $\Gamma = (V, E)$ on an alphabet X we call it *complete* if, for every vertex v and word $u \in X^*$ there exists a path γ in Γ starting from v and labelled by the word u. We denote by $v \odot u$ the unique vertex which is the end of this path γ.

Definition 2. *Let $\Gamma = (V, E, v_1)$ be a 1-graph which is deterministic , complete and such that v_1 is a root. We call* rational structure *on Γ every ($|X|$+2)-tuple of finite automata $\mathcal{S} = (W, M_\epsilon, (M_x)_{x\in X})$ such that*

1. *W is a one-tape deterministic finite automaton on X^* such that $\forall u \in X^*, \exists w \in \mathrm{L}(W), v_1 \odot u = v_1 \odot w$*
2. *M_ϵ is a 2-d.f.a. which is strictly balanced , and $\mathrm{L}(M_\epsilon) = \{(w_1, w_2) \in \mathrm{L}(W) \times \mathrm{L}(W) \mid v_1 \odot w_1 = v_1 \odot w_2\}$*
3. *for every letter $x \in X$, M_x is a 2-d.f.a. which is balanced , and $\mathrm{L}(M_x) = \{(w_1, w_2) \in \mathrm{L}(W) \times \mathrm{L}(W) \mid v_1 \odot w_1 x = v_1 \odot w_2\}$*

Theorem 11. *$\Gamma = (V, E, v_1)$ be a 1-graph which is deterministic and complete. Then Γ has a rational structure $\mathcal{S}$.*
Given a normalized d.p.d.a. such that Γ is the computation 1-graph of $\mathcal{M}$, one can compute such a rational structure $\mathcal{S}$.

4.3 Decision Problems

Let us investigate several decision problems over infinite graphs.
The **isomorphism** problem for **equational** graphs:

INSTANCE: Two equational graphs Γ_1, Γ_2.
QUESTION: Is Γ_1 isomorphic with Γ_2?

This problem has been solved by "model-theoretic" methods in [Cou89,Cou90a] (hence by a method quite different, by nature, from the method used for solving the equivalence problem for dpda in [Sén97c,Sén01,Sti99]). Let us mention that the subproblem where Γ_1, Γ_2 are supposed rooted and deterministic, can be solved as a corollary of the dpda's equivalence problem: it reduces to the property that each graph Γ_i is a homomorphic image of the other and this last problem is solved below by a corollary of the dpda's equivalence problem (theorem 16).

The **isomorphism** problem for **algebraic ordered** trees:

INSTANCE: Two algebraic ordered trees T_1, T_2.
QUESTION: Is T_1 isomorphic with T_2?

Corollary 12. *The isomorphism problem for algebraic ordered trees is decidable.*

This follows from theorem 1 and the reduction given in [Cou83, theorem 5.5.3 p.158].

The **bisimulation** problem for **rooted equational** graphs of **finite out-degree**.

INSTANCE: Two rooted equational graphs of finite out-degree Γ_1, Γ_2.
QUESTION: Is Γ_1 bisimilar to Γ_2?

Theorem 13. *The bisimulation problem for rooted equational graphs of finite out-degree is decidable.*

This kind of problem has been studied for many classes of graphs (or *processes*), see for example [BBK87],[Cau90],[HS91], [CHM93], [GH94], [HJM94],[CHS95], [Cau95],[Sti96],[Jan97],[Sén98a].The case of *rooted equational graphs of finite out-degree* was raised in [Cau95] (see Problem 6.2 of this reference) and is a significant sub case of the problem raised in [Sti96] (as the bisimulation-problem for processes " of type -1"). It is solved in [Sén98a,Sén00a] by a method derived from the one used to solve the equivalence problem for dpda.

The **isomorphism** problem for **algebraic unordered** trees.

INSTANCE: Two algebraic unordered trees T_1, T_2.

QUESTION: Is T_1 isomorphic with T_2?

Theorem 14. *The isomorphism problem for algebraic unordered trees is decidable.*

This result can be proved by a variant of the solution of the bisimulation problem for rooted equational graphs of finite out-degree.
Key idea:
Let us consider two algebraic unordered trees T_1, T_2. Their vertices are labelled on a ranked alphabet (F, ρ). We suppose that these trees have no leaf (one can easily reduce trees in such a normal form, by a transformation which preserves algebraicity and in such a way that isomorphic trees have isomorphic normal forms). Let us consider the two trees T_1', T_2', which are deduced from T_1,T_2 just by removing the labels on the vertices but encoding then the label of a vertex by labels, (we use the alphabet Y described in (11)), on the edges getting out of this vertex. The set of all words labelling a prefix of some branch of T_i' $(1 \leq i \leq 2)$is just the language $Brch(T_i)$, which is recognized by some dpda $\mathcal{A}_i$. Hence T_i' is the infinite unfolding of the computation 1-graph Γ_i of $\mathcal{A}_i$. Note that Γ_i (for $1 \leq i \leq 2$), is a rooted deterministic 1-graph over the alphabet Y and it is equational, by theorem 9. An equivalence relation η on Y is defined by:

$$\forall f, f' \in F, i \in [1, \rho(f)], j \in [1, \rho(f')], ([f, i]\eta[f, j] \Leftrightarrow f = f').$$

Definition 15 *Let Γ Γ' be two deterministic n-graphs over the alphabet Y. $\mathcal{R}$ is a* η-permutative bisimulation *from Γ to Γ' iff*

1. $\mathrm{dom}(\mathcal{R}) = V_\Gamma$, $\mathrm{im}(\mathcal{R}) = V_{\Gamma'}$
2. $\forall i \in [1, n], (v_i, v_i') \in \mathcal{R}$,
3. $\forall (v, v') \in \mathcal{R}$, *there exists a bijection $h_{v,v'}$ from $E(v) = \{(v, y, w) \in E_\Gamma\}$ onto $E(v') = \{(v', y', w') \in E_{\Gamma'}\}$ such that, for every $y \in Y, w \in V_\Gamma$:*
   $h_{v,v'}(v, y, w) = (v', y', w') \Rightarrow (y \eta y')$.

This notion of η-permutative bisimulation is close to the notion of η-bisimulation considered in ([Sén98a, Definition 3.1], [Sén00b, Definition 23]), so that an adaptation of the techniques can be easily done.

The **quotient** problem for **rooted deterministic** equational 1-graphs.

INSTANCE: Two rooted deterministic equational 1-graphs Γ_1, Γ_2.
QUESTION: Does there exist some homomorphism from Γ_1 to Γ_2?

Theorem 16. *The quotient problem for rooted deterministic equational 1-graphs is decidable*

This result can be derived from theorem 11 above and a more general result

Theorem 17. *Let Γ_1, Γ_2 be two rooted deterministic 1-graphs. Let us suppose that Γ_1 is given by an automatic structure $\mathcal{S}$ and that Γ_2 is the computation 1-graph of a given dpda $\mathcal{M}$. One can then decide whether there exists a graph homomorphism $h : \Gamma_1 \to \Gamma_2$.*

Sketch of proof: Let $\mathcal{S} = (W, M_\epsilon, (M_x)_{x\in X})$. For every $a \in X \cup \{\epsilon\}$, let us call a-decomposition, any $2n$-tuple $(u_1, u'_1, u_2, u'_2, \ldots, u_n, u'_n)$ of words such that there exists a computation

$$q_0 \xrightarrow{(u_1,\epsilon)} q_1 \xrightarrow{(\epsilon,u'_1)} q'_1 \xrightarrow{(u_2,\epsilon)} q_2 \xrightarrow{(\epsilon,u'_2)} q'_2 \ldots \xrightarrow{(u_n,\epsilon)} q_n \xrightarrow{(\epsilon,u'_n)} q'_n \in F \tag{16}$$

and

$$u_1 \cdot u_2 \cdots u_n \in X^*\#; \;\; u'_1 \cdot u'_2 \cdots u'_n \in X^*\#.$$

Let $\mathcal{M} =< X, Z, Q, \delta_2, q_0, z_0, F >$. We then define, for every $a \in X \cup \{\epsilon\}$, the languages

$$\mathrm{L}_a = \{u_1 \diamond u'_1 \diamond u_2 \diamond u'_2 \diamond \ldots \diamond u_n \diamond u'_n \diamond\diamond q \;\; t\omega \mid (u_1, u'_1, u_2, u'_2, \ldots, u_n, u'_n) \text{ is some a-decomposition and } q_0 z_0 \xrightarrow[\mathcal{M}]{u_1 u_2 \cdots u_n \#^{-1}} q\omega \text{ where } q \in Q, \omega \in Z^*\}. \tag{17}$$

$$\mathrm{L}'_a = \{u_1 \diamond u'_1 \diamond u_2 \diamond u'_2 \diamond \ldots \diamond u_n \diamond u'_n \diamond\diamond q \;\; t\omega \mid (u_1, u'_1, u_2, u'_2, \ldots, u_n, u'_n) \text{ is some a-decomposition and } q_0 z_0 \xrightarrow[\mathcal{M}]{u'_1 u'_2 \cdots u'_n \#^{-1}} q\omega \text{ where } q \in Q, \omega \in Z^*\}. \tag{18}$$

(Here $\diamond$ is just a new letter not in $X \cup \{\#\}$).
One can check that there exists some homomorphism $h : \Gamma_1 \to \Gamma_2$ iff

1. $\forall u \in X^*, \exists w \in \mathrm{L}(W)$, such that $q_0 z_0 \odot_{\Gamma_2} u = q_0 z_0 \odot_{\Gamma_2} w$
2. for every $a \in X \cup \{\epsilon\}$, $\mathrm{L}_a = \mathrm{L}'_a$.

Point 1 reduces to test the equality between the two rational languages over the alphabet $Q \cup Z$:

$$\{q\omega \in QZ^* \mid \exists u \in X^*, q_0 z_0 \xrightarrow[\mathcal{M}]{u} q\omega\}, \{q\omega \in QZ^* \mid \exists w \in \mathrm{L}(W), q_0 z_0 \xrightarrow[\mathcal{M}]{w} q\omega\}$$

which can be effectively done.
Point 2 amounts to test a finite number of dpda equivalences, which can be effectively done, by theorem 1. □

5 Thue-Systems

5.1 Abstract Rewriting Systems

Let E be some set and $\longrightarrow$ some binary relation over E .We shall call $\longrightarrow$ the *direct reduction*.We shall use the notations $\xrightarrow{i}$ for every integer $(i \geq 0)$, and $\xrightarrow{*}, \xrightarrow{+}$ in the usual way (see [Hue80]). By $\longleftrightarrow$, we denote the relation

$\longrightarrow \cup \longleftarrow$. The three relations $\xrightarrow{*}, \xleftarrow{*}$ and $\xleftrightarrow{*}$ are respectively the *reduction*, *derivation* and the *equivalence* generated by $\longrightarrow$.

We use the notions of *confluent* relation and *noetherian* relation in their usual meaning ([Hue80] or [DJ91, §4, p.266-269]).

An element $e \in E$ is said to be *irreducible*(resp. *reducible*) modulo ($\longrightarrow$) iff there exists no (resp. some) $e' \in E$ such that $e \longrightarrow e'$. By $\mathrm{Irr}(\longrightarrow)$ we denote the set of all the elements of E which are irreducible modulo ($\longrightarrow$).

Given some subset A of E, we use the following notation:

$$< A >_{\xleftarrow{*}} = \{e \in E \mid \exists a \in A, a \xleftarrow{*} e\}, \quad [A]_{\xleftrightarrow{*}} = \{e \in E \mid \exists a \in A, a \xleftrightarrow{*} e\}$$

5.2 Semi-Thue Systems

We call a subset $S \subseteq X^* \times X^*$ a *semi-Thue system* over X. By $\longrightarrow_S$ we denote the binary relation defined by: $\forall f, g \in X^*$,

$$f \longrightarrow_S g \text{ iff there exists } (u, v) \in S, \alpha, \beta \in X^* \text{ such that } f = \alpha u \beta, g = \alpha v \beta.$$

$\longrightarrow_S$ is the one-step reduction generated by S. All the definitions and notation defined in the §"abstract rewriting systems" apply to the binary relation $\longrightarrow_S$. Let us give now additional notions, notation and results which are specific to semi-Thue systems.

We use now the notation $\mathrm{Irr}(S)$ for $\mathrm{Irr}(\longrightarrow_S)$. We define the *size* of S as $\|S\| = \Sigma_{(u,v)\in S}|u| + |v|$. A system S is said *strictly length-reducing* (strict, for short) iff, $\forall (u, v) \in S, |u| > |v|$.

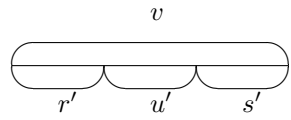
Definition 3. *Let us consider the following conditions on the rules of a semi-Thue system S:*

C1 : for every $(u, v), (u', v') \in S$ *and every* $r', s' \in X^*$
$v = r'u's' \Longrightarrow | \ s' \ | = | \ r' \ | = 0$
C2 : for every $(u, v), (u', v') \in S$ *and every* $r, s' \in X^*$
$rv = u's' \Longrightarrow | \ s' \ | = 0$ *or* $| \ s' \ | \geq | \ v \ |$
C3 : for every $(u, v), (u', v') \in S$ *and every* $r', s \in X^*$
$vs = r'u' \Longrightarrow | \ r' \ | = 0$ *or* $| \ r' \ | \geq | \ v \ |$
S *is said* special *iff* $\forall (u, v) \in S, | \ v \ | = 0$
S *is said* monadic *iff* $\forall (u, v) \in S, | \ v \ | \leq 1$
S *is said* basic *iff it fulfills C1, C2 and C3*
S *is said* left-basic *iff it fulfills C1 and C2.*

Each condition $C_i (i \in [1, 3])$ consists in the prohibition of some superposition configuration for two redexes $(r, u, v), (r', u', v')$ of S.

Condition C1 : C1 expresses the prohibition of the following configuration:
$v = r'u's'$ where $| \ u' \ | < | \ v \ |$

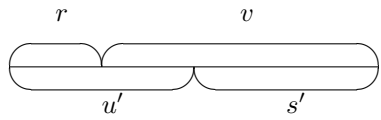
schema 1:



In other words, a righthand side of rule may not strictly embed any lefthand side of rule.

Condition C2 : C2 expresses the prohibition of the following configuration: $rv = u's'$ where $0 <| s' |<| v |$

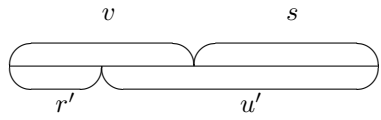
schema 2:



In other words, a righthand side of rule may not be "strictly overlapped on the left" by any lefthand side of rule.

Condition C3 : C3 expresses the prohibition of the following configuration: $vs = r'u'$ where $0 <| r' |<| v |$.

schema 3:



In other words, a right-hand side of rule may not be "strictly overlapped on the right" by any left-hand side of rule.
These definitions appeared in [Niv70,Coc71,But73,Sak79]. We refer the reader to [DJ91,BO93] for more information on rewriting systems and [Sén94] for links with formal language theory.

5.3 Decision Problems

It is well-known that the *confluence* property, for a semi-Thue system S, can be decided, provided that the relation $\longrightarrow_S$ is noetherian. Let us consider the following "weak confluence" property:

$$< f >_{\overset{*}{\longleftarrow}_S} = [f]_{\overset{*}{\longleftrightarrow}_S}. \tag{19}$$

A system S fulfilling (19) for some irreducible word f, is said *confluent over the class of* f. Such a property deserves some interest because,

- the language $< f >_{\overset{*}{\longleftarrow}_S}$ can be recognized in linear time, as soon as S is strict and finite (this motivation remains valid even for some graph rewriting systems, see [ACPS93]).
- in the case where $X^*/\overset{*}{\longleftrightarrow}_S$ is a group, the confluence property over the class of the empty word, ε, is sufficient to ensure decidability of the word-problem.

Let us call **Class Confluence** Problem (CCP for short), the following decision-problem:

INSTANCE A finite alphabet X, a noetherian semi-Thue system S over X and a word $f \in X^*$, which is irreducible modulo S.
QUESTION Is S confluent on the class $[f]_{\overset{*}{\longleftrightarrow}_S}$?

Several works have been devoted to the decidability/complexity of problems similar to CCP ([Sén85], [ABS87], [Ott87], [Nar90], [Sén90], [OZ91], [Zha91], [MNO91], [Ott92a], [Ott92b], [Zha92], [GG93], [MNOZ93], [Eng94], [Sén98b]). In particular it is known that CCP becomes

- *undecidable* for strict, finite, semi-Thue systems ([Ott92b])
- decidable in *polynomial time* for strict, finite, basic semi-Thue systems ([Sén98b]).

Corollary 18. *The Class Confluence Problem is decidable for strict, finite, left-basic semi-Thue systems.*

This corollary follows from theorem 1 and the reduction given in [Sén90, theorem 5.17]. In the context of the two above complexities obtained for general semi-Thue systems and for basic semi-Thue systems, it is a natural challenge to determine the complexity of the CCP for strict, finite, left-basic semi-Thue systems. It is not even known, at the moment, whether this problem is *primitive recursive* or not (from this point of view, its status is the same as for the dpda's equivalence problem, since the reduction given in [Sén90] is in DEXP-time and a converse reduction, in DEXP-time too, is given).

Acknowledgement

The author thanks the organizers of MCU2001 for their invitation.

References

ABS87. J.M. Autebert, L. Boasson, and G. Sénizergues. Groups and NTS languages. *JCSS vol.35, no2*, pages 243–267, 1987.

ACPS93. S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *J. ACM 40*, pages 1134–1164, 1993.

Bau92. M. Bauderon. Infinite hypergraph II, systems of recursive equations. *TCS 103*, pages 165–190, 1992.

BBK87. J. Baeten, J. Bergstra, and J. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In *Proceedings of PARLE 87*, pages 94–111. LNCS 259, 1987.

BG00. A. Blumensath and E. Gräedel. Automatic structures. In *Proceedings LICS*, pages 105–118. IEEE Computer Society Press, 2000.

BO93. R.V. Book and F. Otto. *String Rewriting Systems.Texts and monographs in Computer Science*. Springer-Verlag, 1993.

But73. P. Butzbach. Une famille de congruences de Thue pour lesquelles l'équivalence est décidable. In *Proceedings 1rst ICALP*, pages 3–12. LNCS, 1973.

Cau90. D. Caucal. Graphes canoniques des graphes algébriques. *RAIRO, Informatique Théorique et Applications, 24(4)*, pages 339–352, 1990.

Cau95. D. Caucal. Bisimulation of context-free grammars and of pushdown automata. In *Modal Logic and process algebra*, pages 85–106. CSLI Lectures Notes, vol. 53, 1995.

CHM93. S. Christensen, Y. Hirshfeld, and F. Moller. Bisimulation is decidable for basic parallel processes. *LNCS 715, Springer*, pages 143–157, 1993.

CHS95. S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation 121*, pages 143–148, 1995.

Coc71. Y. Cochet. Sur l'algébricité des classes de certaines congruences définies sur le monoide libre. *Thèse de l'université de Rennes*, 1971.

Cou78a. B. Courcelle. A representation of trees by languages,I. *Theoretical Computer Science 6*, pages 255–279, 1978.

Cou78b. B. Courcelle. A representation of trees by languages,II. *Theoretical Computer Science 7*, pages 25–55, 1978.

Cou83. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science 25*, pages 95–169, 1983.

Cou89. B. Courcelle. The monadic second-order logic of graphs ii: infinite graphs of bounded width. *Math. Systems Theory 21*, pages 187–221, 1989.

Cou90a. B. Courcelle. The monadic second-order logic of graphs iv: definability properties of equational graphs. *Annals of Pure and Applied Logic 49*, pages 193–255, 1990.

Cou90b. B. Courcelle. Recursive applicative program schemes. In *Handbook of Theoretical Computer Science, edited by J. Van Leeuwen*, pages 461–490. Elsevier, 1990.

CS99. A. Carbone and S. Semmes. A graphic apology for symmetry and implicitness. *Preprint*, 1999.

DJ91. N. Dershowitz and J.P. Jouannaud. Rewrite systems. In *Handbook of theoretical computer science, vol.B, Chapter 2*, pages 243–320. Elsevier, 1991.

ECH+92. D.B.A. Epstein, J.W. Cannon, D.F. Holt, S.V.F. Levy, M.S. Paterson, and W.P. Thurston. *Word processing in groups*. Jones and Bartlett, 1992.

Eng94. J. Engelfriet. Deciding the NTS-property of context-free grammars. In *Important Results and Trends in Theoretical Computer Science, (Colloquium in honor of Aarto Salomaa)*, pages 124–130. Springer-Verlag,LNCS nr 812, 1994.

Fri77. E.P. Friedman. Equivalence problems for deterministic context-free languages and monadic recursion schemes. *Journal of Computer and System Sciences 14*, pages 344–359, 1977.

Gal81. J.H. Gallier. Dpda's in 'atomic normal form' and applications to equivalence problems. *Theoretical Computer Science 14*, pages 155–186, 1981.

GG66. S. Ginsburg and S. Greibach. Deterministic context-free languages. *Information and Control*, pages 620–648, 1966.

GG93. P. Grosset-Grange. Décidabilité de la confluence partielle d'un système semi-Thuéien rationnel. *Mémoire de DEA de l'université de Bordeaux 1*, pages 1–21, 1993.

GH94. J. Groote and H. Hüttel. Undecidable equivalences for basic process algebra. *Information and Computation 115*, pages 354–371, 1994.

GL73. S.J. Garland and D.C. Luckham. Program schemes, recursion schemes, and formal languages. *Journal of Computer and System Sciences 7*, pages 119–160, 1973.

Gue81. I. Guessarian. *Algebraic Semantics*. Lecture Notes in Computer Science, vol. 99, 1981.

HJM94. Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding equivalence of normed context-free processes. In *Proceedings of FOCS'94*, pages 623–631. IEEE, 1994.

HS91. H. Hüttel and C. Stirling. Actions speak louder than words: Proving bisimilarity for context-free processes. In *LICS'91*, pages 376–385. IEEE, 1991.

Hue80. G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *JACM vol. 27 no 4*, pages 797–821, 1980.

Ian60. I.I. Ianov. The logical schemes of algorithms. *Problems of Cybernetics (USSR) 1*, pages 82–140, 1960.

Jan97. P. Jancar. Bisimulation is decidable for one-counter processes. In *Proceedings ICALP 97*, pages 549–559. Springer Verlag, 1997.

Kap69. D.M. Kaplan. Regular expressions and the equivalence of programs. *Journal of Computer and Systems Sciences 3*, pages 361–386, 1969.

KN95. B. Khoussainov and A. Nerode. Automatic presentations of structures. In *Logic and Computational Complexity, Indianapolis,94*, pages 367–392. L.N.C.S nr 960, 1995.

LPP70. D.C. Luckham, D.M. Park, and M.S. Paterson. On formalised computer programs. *Journal of Computer and Systems Sciences 4*, pages 220–249, 1970.

Ly00a. O. Ly. Automatic graphs and D0L-sequences of finite graphs. *Preprint, submitted to JCSS*, pages 1–33, 2000.

Ly00b. O. Ly. Automatic graphs and graph D0L-systems. In *Proceedings MFCS'2000*, pages 539–548. Springer, LNCS No 1893, 2000.

Mat95. Y.V. Matiyasevich. Personal communication. 1995.

Mil70. R. Milner. Equivalences on program schemes. *Journal of Computer and Systems Sciences 4*, pages 205–219, 1970.

Mil89. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

MNO91. K. Madlener, P. Narendran, and F. Otto. A specialized completion procedure for monadic string-rewriting systems presenting groups. In *Proceedings 18th ICALP*, pages 279–290. Springer, LNCS No 510, 1991.

MNOZ93. K. Madlener, P. Narendran, F. Otto, and L. Zhang. On weakly confluent monadic string-rewriting systems. *T.C.S. 113*, pages 119–165, 1993.

Mor00. C. Morvan. Rational graphs. In *Proceedings STACS'2000*. LNCS, 2000.

Nar90. P. Narendran. It is decidable whether a monadic Thue system is canonical over a regular set. *Math. Systems Theory 23*, pages 245–254, 1990.

Niv70. M. Nivat. On some families of languages related to the Dyck language. *2nd Annual Symposium on Theory of Computing*, 1970.

Niv75. M. Nivat. *On the interpretation of recursive polyadic program schemes.* Symposia Mathematica, vol. 15, Academic press, New-York, 1975.

Ott87. F. Otto. On deciding the confluence of a finite string-rewriting system on a given congruence class. *JCSS 35*, pages 285–310, 1987.

Ott92a. F. Otto. Completing a finite special string-rewriting system on the congruence class of the empty word. *Applicable Algebra in Engeneering Comm.Comput. 2*, pages 257–274, 1992.

Ott92b. F. Otto. The problem of deciding confluence on a given congruence class is tractable for finite special string-rewriting systems. *Math. Systems Theory 25*, pages 241–251, 1992.

OZ91. F. Otto and L. Zhang. Decision problems for finite special string-rewriting systems that are confluent on some congruence class. *Acta Informatica 28*, pages 477–510, 1991.

Par81. D. Park. Concurrency and automata on infinite sequences. *LNCS 104*, pages 167–183, 1981.

Pat67. M.S. Paterson. Equivalence problems in a model of computation. *Ph. D. Thesis, University of Cambridge, England*, 1967.

Pel97. L. Pelecq. *Isomorphismes et automorphismes des graphes context-free, équationnels et automatiques.* Thèse de doctorat, Université Bordeaux I, Juin 1997.

Ros75. B.K. Rosen. Program equivalence and context-free grammars. *Journal of Computer and System Sciences 11*, pages 358–374, 1975.

Rut64. J. Rutledge. On Ianov's program schemata. *Journal of the Association for Computing Machinery*, pages 1–9, 1964.

Sak79. J. Sakarovitch. Syntaxe des langages de Chomsky, essai sur le déterminisme. *Thèse de doctorat d'état de l'université Paris VII*, pages 1–175, 1979.

Sén85. G. Sénizergues. The equivalence and inclusion problems for NTS languages. *J. Comput. System Sci. 31(3)*, pages 303–331, 1985.

Sén90. G. Sénizergues. Some decision problems about controlled rewriting systems. *TCS 71*, pages 281–346, 1990.

Sén92. G. Sénizergues. Definability in weak monadic second-order logic of some infinite graphs. In *Dagstuhl seminar on Automata theory: Infinite computations. Wadern, Germany*, volume 28, pages 16–16, 1992.

Sén94. G. Sénizergues. Formal languages and word-rewriting. In *Term Rewriting, Advanced Course*, pages 75–94. Springer, LNCS 909, edited by H. Comon and J.P.Jouannaud, 1994.

Sén97a. G. Sénizergues. L(A) = L(B)? In *Proceedings INFINITY 97*, pages 1–26. Electronic Notes in Theoretical Computer Science 9, URL: http://www.elsevier.nl/locate/entcs/volume9.html, 1997.

Sén97b. G. Sénizergues. L(A) = L(B)? Technical report, LaBRI, Université Bordeaux I, report nr1161-97, 1997. Pages 1-71.

Sén97c. G. Sénizergues. The Equivalence Problem for Deterministic Pushdown Automata is Decidable. In *Proceedings ICALP 97*, pages 671–681. Springer, LNCS 1256, 1997.

Sén98a. G. Sénizergues. Decidability of bisimulation equivalence for equational graphs of finite out-degree. In Rajeev Motwani, editor, *Proceedings FOCS'98*, pages 120–129. IEEE Computer Society Press, 1998.

Sén98b. G. Sénizergues. A polynomial algorithm testing partial confluence of basic semi-Thue systems. *Theoretical Computer Science 192*, pages 55–75, 1998.

Sén00a. G. Sénizergues. Complete Formal Systems for Equivalence Problems. *Theoretical Computer Science*, 231:309–334, 2000.

Sén00b. G. Sénizergues. The bisimulation problem for equational graphs of finite out-degree. *submitted to SIAM Journal on Computing; can be accessed at URL:http://arXiv.org/abs/cs/0008018*, pages 1–98, 2000.

Sén01. G. Sénizergues. L(A) = L(B)? decidability results from complete formal systems. *Theoretical Computer Science*, 251:1–166, 2001.

Sol78. M. Solomon. Type definitions with parameters. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 31–38, Tucson, Arizona, 1978. ACM SIGACT-SIGPLAN. Extended abstract.

Sti96. C. Stirling. Decidability of bisimulation equivalence for normed pushdown processes. In *Proceedings CONCUR 96*, pages 217–232. Springer-Verlag, LNCS 1119, 1996.

Sti99. C. Stirling. Decidability of dpda's equivalence. Technical report, Edinburgh ECS-LFCS-99-411, 1999. Pages 1-25, accepted for publication in TCS.

Zha91. L. Zhang. Weak confluence is tractable for finite string-rewriting systems. *preprint*, pages 1–10, 1991.

Zha92. L. Zhang. The pre-NTS property is undecidable for context-free grammars. *IPL 44*, pages 181–184, 1992.

The Equivalence Problem for Computational Models: Decidable and Undecidable Cases

Vladimir A. Zakharov

Faculty of Computational Mathematics and Cybernetics,
Moscow State University, Moscow, RU-119899, Russia
zakh@cs.msu.su

Abstract. This paper presents a survey of fundamental concepts and main results in studying the equivalence problem for computer programs. We introduce some of the most-used models of computer programs, give a brief overview of the attempts to refine the boarder between decidable and undecidable cases of the equivalence problem for these models, and discuss the techniques for proving the decidability of the equivalence problem.

Informally, the equivalence problem is to find out whether two given programs have the same behavior. By picking various formal definitions of the terms "program" and "behavior" we get numerous variants of this problem. The study of the equivalence problem for computational models is always of basic interest in computer science beginning with the original works on the development of the formal concept of computer program [24,45,34,9,11]. Tackling the equivalence problem we comprehend to what extent the very nature of computations can be conceived by means of formal methods and how much specific changes in the structure of a computer program affect its behavior. The understanding of relationships between the syntactic and semantic components of programs is very important for specification, verification and optimization of programs, partial computations, reusing of programs, etc. That is why this problem significantly influences both the theory and the practice of computer science and software engineering.

The decidability of the equivalence problem essentially depends on the expressive power of a computational model and the exact meaning of the term "the same behavior". Two fundamental results of 50th governed the advancement in studies of the equivalence problem for computer programs.

When programs under consideration are deterministic, it is quite reasonable to assume that two programs have the same behaviour if they compute the same function, i.e. for every valid input they output identical results (if any). A mapping which associates each program π with the recursive function f_π computed by π is called *enumeration* of recursive functions. If a programming system PS has an effective interpreter that can simulate every program by presenting its description as a part of the input, then enumeration is called *computable*. If, moreover, a programming system PS is such that any other programming system PS' corresponding to computable enumeration can be effectively translated

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 133–152, 2001.

into PS then the enumeration specified by PS is called *acceptable* (see [47,54]). In 1953 H.G.Rice [46] proved that if a programming system PS corresponds to an acceptable enumeration of recursive functions, then any non-trivial property of computer programs which refers only to functions computed by programs is undecidable. This implies the undecidability of the functional equivalence of computer programs for every natural universal programming system.

On the other hand, in 1956 A.A.Lyapunov and Y.I.Yanov [24] introduced a propositional model of sequential programs (Yanov schemata) and set up the equivalence problem for this model. Yanov scheme may be thought of (see [11]) as a finite transition system whose nodes are labelled with statements $A_1, \ldots, A_n$ and basic propositions $p_1, \ldots, p_m$. Each statement A is associated with a set of basic proposition $Sh(A)$. Nodes marked with statements are called transformers; each transformer has a single outgoing arc. Nodes marked with propositions are called recognizor; each recognizor has two outgoing arcs labelled with $\top$ and $\bot$. Transformers are understood as abstractions of program statements, whereas basic propositions stand for logic conditions. Two specific nodes — **entry** and **exit** are distinguished. A run of a Yanov scheme π starts from the **entry** given some initial evaluation δ^0 of basic propositions. When a run passes via a transformer labelled with A, the statement A is executed by changing arbitrary the values of propositions from $Sh(A)$. When a run passes via a recognizor labelled with basic proposition p, it checks the current value of p and follows the arc marked with this value. A run terminates as soon as it reaches the **exit**. A determinant of a run r is a sequence of pairs $(\delta^0, A^1), (\delta^1, A^2), \ldots, (\delta^{n-1}, A^n), \ldots$, where $A^1, A^2, \ldots, A^n, \ldots$ is a sequence of statements executed during this run, and $\delta^i,\ i \geq 1$ is the evaluation of basic proposition after executing A^i. A determinant $det(\pi)$ of a Yanov scheme π is said to be a set of determinants of all possible terminated runs of π. Two schemata π_1 and π_2 are called equivalent if $det(\pi_1) = det(\pi_2)$. In 1957 Yanov [58] found a complete axiomatization (equational calculus) for the equivalence relation thus defined and proved that the equivalence problem for this computational model is decidable.

Both results, Rice's theorem and the decidability of equivalence problem for Yanov schemata, were the pioneering steps from the opposite sides towards the boarder between decidability and undecidability of the equivalence problem for computational models. Since that time the refinement of this boarder has become the topic for a large body of research. In succeeding years a wide variety of devices defining computations, languages and translations were introduced [45,38,9,32,35]. Every time when an all-new model of computation made its appearance, the equivalence problem for this model challenged researchers. Considerable effort devoted to this problem yielded an amazing amount of results and techniques that substantially extend the capability of formal methods in computer science.

Now it is hardly possible to cover in a single survey all main results and approaches to the equivalence problem for the most important models of computation. In this paper we consider in some details the current state of art in studying the equivalence problem for two computational models — propositional sequen-

tial programs and first-order sequential programs. An overview of the techniques for proving the decidability of language equivalence and relational equivalence for devices defining languages and translations (relations) can be found in [8]. Since the publishing of this paper, some new results has been obtained [13,51] that substantially revised our understanding of decidability/undecidability frontier for the equivalence problem. A broad spectrum of results on the bisimilation equivalence problem for automata and process algebras is discussed in much details in [14].

1 Propositional Sequential Programs

In this section we introduce the concept of a propositional sequential program (PSP), its syntax, and semantics and discuss the principal results on the equivalence problem for this model of computations.

1.1 Syntax of PSP

Fix two finite alphabets $\mathcal{A} = \{a^1, \ldots, a^N\}$ and $\mathcal{P} = \{p_1, \ldots, p_M\}$.

The elements of $\mathcal{A}$ are called *basic actions*. Intuitively, the basic actions stand for the elementary program statements such as assignment statements and procedure calls. A finite sequence of basic actions is called a *term*. The set of all terms is denoted by $\mathcal{A}^*$. We write λ for the empty term, $|h|$ for the length of a term h, and hg for the concatenation of terms h and g.

The symbols of $\mathcal{P}$ are called *basic propositions*. We assume that basic propositions denote the primitive relations on program data. Each basic proposition may be evaluated either by $\perp$ (falsehood) or by $\top$ (true). A binary M-tuple $\delta = \langle d_1, \ldots, d_M \rangle$ of truth-values of all basic propositions is called a *condition*. We write $\mathcal{C}$ for the set of all conditions.

A *propositional sequential program* (PsP) over alphabets $\mathcal{A}$, $\mathcal{P}$ is a tuple $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, \mathbf{loop}, B, T \rangle$, where

- V is a finite set of *program nodes*;
- **entry** is an *initial node*, $\mathbf{entry} \in V$,
- **exit** is a *terminal node*, $\mathbf{exit} \in V$,
- **loop** is a *dead node*, $\mathbf{loop} \in V$,
- $B : V \to \mathcal{A}^*$ is a *binding function*, associating every node with some term so that $B(\mathbf{entry}) = B(\mathbf{exit}) = B(\mathbf{loop}) = \lambda$;
- $T : (V - \{\mathbf{exit}\}) \times \mathcal{C} \to V$ is a total *transition function* such that $T(\mathbf{loop}, \delta) = \mathbf{loop}$ for every δ from $\mathcal{C}$.

In essence, a PSP may be thought of as a finite-state labelled transition system representing the control structure of a sequential program. We extend a binding function B to the sequences of program nodes by assuming $B(v_1, v_2, \ldots, v_k) = B(v_1)B(v_2)\ldots B(v_k)$. By the size $|\pi|$ of a given PSP π we mean the number $|V|$ of its internal nodes.

1.2 Semantics of PSP

The semantics of PSP is defined by means of dynamic Kripke structures (frames and models) (see [12]).

A *dynamic deterministic frame* (or simply a *frame*) over alphabet $\mathcal{A}$ is a triple $\mathcal{F} = \langle S, s_0, R\rangle$, where

- S is a non-empty set of *data states*,
- s_0 is an *initial state*, $s_0 \in S$,
- $R: \ S \times \mathcal{A} \ \rightarrow \ S$ is an *updating function*.

$R(s, a)$ is interpreted as a result of the application of an action a to a data state s. It is assumed that each basic action $a \in \mathcal{A}$ transforms deterministicaly one state into another; therefore, the dynamic frames under considerations are both functional and serial relative to every action $a \in \mathcal{A}$.

An updating function R can be naturally extended to the set $\mathcal{A}^*$ as follows

$$R^*(s, \lambda) = s, \ R^*(s, ha) = R(R^*(s, h), a).$$

We say that a state s'' is *reachable* from a state s' ($s' \preceq_{\mathcal{F}} s''$ in symbols) if $s'' = R^*(s', h)$ for some $h \in \mathcal{A}^*$. Denote by $[h]_{\mathcal{F}}$ the state $s = R^*(s_0, h)$ which is reachable from the initial state by means of $\mathcal{A}$-sequence h. As usual, the subscript $\mathcal{F}$ will be omitted when the frame is understood. We will deal only with data states that are reachable from the initial state. Therefore, we may assume, without loss of generality, that every state $s \in S$ is reachable from the initial state s_0, i.e. $S = \{[h]: \ h \in \mathcal{A}^*\}$.

A frame $\mathcal{F}_s = \langle S', s, R'\rangle$ is called a *subframe* of a frame $\mathcal{F} = \langle S, s_0, R\rangle$ generated by a state $s \in S$ if $S' = \{R^*(s, h): \ h \in \mathcal{A}^*\}$ and R' is a restriction of R to S'. We say that a frame $\mathcal{F}$ is

- *semigroup* if $\mathcal{F}$ can be mapped homomorphically onto every subframe $\mathcal{F}_s$,
- *homogeneous* if $\mathcal{F}$ is isomorphic to every subframe $\mathcal{F}_s$,
- *ordered* if $\preceq$ is a partial order on the set of data states S,
- *universal* if $[h] = [g]$ implies $h = g$ for every pair h, g of $\mathcal{A}$-sequences.

Taking the initial state $s_0 = [\lambda]$ for the unit, one may regard a semigroup frame $\mathcal{F}$ as a finitely generated monoid $\mathcal{F} = \langle S, *\rangle$ such that $[h] * [g] = [hg]$. Clearly, the universal frame $\mathcal{U}$ corresponds to the free monoid generated by $\mathcal{A}$. Consequently, an ordered semigroup frame is associated with a monoid whose unit $[\lambda]$ is irresolvable, i.e. $[\lambda] = [gh]$ implies $g = h = \lambda$, whereas a semigroup corresponding to a homogeneous frame is a left-contracted monoid, i.e. $[gh'] = [gh'']$ implies $[h'] = [h'']$.

A *dynamic deterministic model* (or simply a *model*) over alphabets $\mathcal{A}$, $\mathcal{P}$ is a pair $M = \langle \mathcal{F}, \xi\rangle$ such that

- $\mathcal{F} = \langle S, s_0, R\rangle$ is a frame over $\mathcal{A}$,
- $\xi: \ S \ \rightarrow \ \mathcal{C}$ is a *valuation function*, indicating truth-values of basic propositions at every data state.

Let $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, \mathbf{loop}, B, T\rangle$ be some PSP and $M = \langle \mathcal{F}, \xi\rangle$ be a model based on the frame $\mathcal{F} = \langle S, s_0, R\rangle$. A finite or infinite sequence of pairs

$$r = (v_0, \delta_0), (v_1, \delta_1), \ldots, (v_m, \delta_m), \ldots \tag{1}$$

where $v_i \in V,\ \delta_i \in \mathcal{C},\ i \geq 0$, is called a *run* of π on M if (1) meets the following requirements:

1. $v_0 = \mathbf{entry}$;
2. $v_{i+1} = T(v_i, \delta_i)$ for every $i,\ i \geq 0$;
3. $\delta_i = \xi([B(v_0, v_1, \ldots, v_i)])$ for every $i,\ i \geq 0$;
4. (1) is a finite sequence iff $T(v_m, \delta_m) = \mathbf{exit}$ for some $m \geq 0$.

When a run r ends with a pair (v_m, δ_m) as its last element, we say that r *terminates*, having the data state $s_m = [B(v_0, v_1, \ldots, v_m)]$ as the *result*. Otherwise, when r is an infinite sequence we say that it *loops* and has no result. Since all PSPs and frames under consideration are deterministic, every program π has a unique run $\pi(M)$ on a given model M. We write $[\pi(M)]$ for the result of $\pi(M)$, assuming $[\pi(M)]$ is undefined when the run loops.

1.3 The Equivalence Problem for PSPs

Let π' and π'' be some PSPs, M a model, $\mathcal{M}$ be a set of models, and $\mathcal{F}$ a frame. Then π' and π'' are called

- *equivalent on* M ($\pi' \sim_M \pi''$ in symbols) if $[\pi'(M)] = [\pi''(M)]$, i.e. either both runs $r(\pi', M)$ and $r(\pi'', M)$ loop or both of them terminate with the same data state as their results,
- *equivalent on* $\mathcal{M}$ ($\pi' \sim_{\mathcal{M}} \pi''$ in symbols) if $\pi' \sim_M \pi''$ holds for every $M \in \mathcal{M}$,
- *equivalent on* $\mathcal{F}$ ($\pi' \sim_{\mathcal{F}} \pi''$ in symbols) if π' and π'' are equivalent on every model $M = \langle \mathcal{F}, \xi\rangle$ based on $\mathcal{F}$.

For a given set of models $\mathcal{M}$ (a frame $\mathcal{F}$) the *equivalence problem* w.r.t. $\mathcal{M}$ ($\mathcal{F}$) is to check for an arbitrary pair π_1, π_2 of PSPs whether $\pi' \sim_{\mathcal{M}} \pi''$ ($\pi' \sim_{\mathcal{F}} \pi''$) holds.

1.4 Decidable and Undecidable Cases of the Equivalence Problem for PSPs

Since we are interested in the algorithmic aspects of the equivalence problem, it is assumed that frames and models we deal with are effectively characterized in logic or algebraic terms (say, by means of dynamic logic formulae or semigroup identities). It is also clear that the undecidability of the identity problem "$[h'] = [h'']$?" for a frame $\mathcal{F}$ implies the undecidability of the equivalence problem for PSP on $\mathcal{F}$. Therefore, we direct our attention only to those semantics (frames and sets of models) that have rather simple and natural arrangement.

Yanov schemata. The computational model of Yanov schemata [24,58,11] was the first attempt to provide a precise mathematical basis for the common activities involved in reasoning about computer program. It is also the first case of computer program formalization whose equivalence problem was proved to be decidable. In the framework of our concept Yanov schemata may be thought of as PSPs whose semantics is based on the set of dynamic models characterized by PDL axioms

$$[a_i]p_j \equiv p_j$$

for every basic action a_i and basic proposition p_j which is not in the shift $Sh(a_i)$. In fact, to check the equivalence of Yanov schemata it suffices to consider their behaviour only on the universal frame $\mathcal{U}$ which is based on the free monoid generated by $\mathcal{A}$. This demonstrates close relationships between Yanov schemata and finite automata [45]. It was revealed in [48] and opened up fresh opportunities for applications of formal methods in computer program analysis.

Ordered frames. When a set of states S of a frame $\mathcal{F}$ is partially ordered w.r.t. $\preceq_{\mathcal{F}}$, this means that no actions in $\mathcal{A}$ are invertible and, hence, PSP's computations always progress. First results on the equivalence problem for PSPs on ordered frames were obtained by Letichevsky.

Theorem 1 ([17]). *Suppose $\mathcal{F}$ is a homogeneous ordered frame such that the identity problem "$[g] = [h]$?" is decidable in time $\psi(n)$. Then the equivalence problem "$\pi_1 \sim_{\mathcal{F}} \pi_2$?" is decidable in time $O(2^n\psi(n))$.*

In many cases the complexity of decision procedures can be improved by taking into account some specific properties of the frames.

Let $\mathcal{F} = \langle S, s_0, R\rangle$ be a semigroup frame. Considering it as a monoid, we write $\mathcal{F} \times \mathcal{F}$ for the direct product of the monoids. Suppose W is a finitely generated monoid, U is a submonoid of W, and w^+, w^* are two distinguished elements in W. Denote by $\circ$ and e a binary operation on W and the unit of W respectively. We say that the quadruple $K = \langle W, U, w^+, w^*\rangle$ is a k_0-*criteria system* for the semigroup frame $\mathcal{F}$, where k_0 is some positive integer, if K and $\mathcal{F}$ meet the following conditions:

1. there exists a homomorphism φ of $\mathcal{F} \times \mathcal{F}$ in U such that

$$[h] = [g] \Leftrightarrow w^+ \circ \varphi(\langle [g], [h]\rangle) \circ w^* = e$$

   holds for every pair g, h in $\mathcal{A}^*$,
2. for every element w in the coset $U \circ w^*$ the equation $y \circ w = e$ has at most k_0 pairwise different solutions y in the coset $w^+ \circ U$.

Theorem 2 ([59]). *Suppose an ordered semigroup frame $\mathcal{F}$ satisfies the following requirements:*

1. *the reachability problem "$[g] \preceq_{\mathcal{F}} [h]$?" is decidable in time $t_1(m)$, where $m = \max(|g|, |h|)$;*

2. *$\mathcal{F}$ has a k_0-criteria system $K = \langle W, U, w^+, w^* \rangle$ such that the identity problem "$w_1 = w_2$?" in W is decidable in time $t_2(m)$, where $m = \max(|w_1|, |w_2|)$.*

Then the equivalence problem "$\pi_1 \sim_{\mathcal{M}} \pi_2$?" is decidable in time $O(n^4(t_1(n^2) + t_2(cn^2) + \log n))$, where $n = \max(|\pi_1|, |\pi_2|)$.

By finding appropriate criterial systems one may construct uniformly polynomial-time decision procedures for the variety of well-known frames. Consider, for example, a partially commutative monoid $\mathcal{F}_I$ induced by some independency relation I [33] on the set of basic actions,

Corollary 1 ([43]). *The equivalence problem for PSP on the partially commutative frame $\mathcal{F}_I$ is decidable in time $O(n^2 \log n)$*

Some results on the equivalence problem for PSPs on ordered frames which are not semigroup were obtained in [61].

Let $\mathcal{F}$ be an arbitrary frame. For every pair of states $\langle s', s'' \rangle$ denote by $I(s_1, s_2)$ the set of pairs of terms $\langle h_1, h_2 \rangle$, such that $R^*(s_1, h_1) = R^*(s_2, h_2)$. Two pairs $\langle s'_1, s'_2 \rangle$ and $\langle s''_1, s''_2 \rangle$ are said to be *similar* ($\langle s'_1, s'_2 \rangle \approx \langle s''_1, s''_2 \rangle$ in symbols) if $I(s'_1, s'_2) = I(s''_1, s''_2)$ holds. We denote by $\langle s'_1, s'_2 \rangle / \approx$ the similarity class containing a pair $\langle s'_1, s'_2 \rangle$.

Theorem 3 ([61]). *Suppose that an ordered frame $\mathcal{F}$ satisfies the following requirements:*

1. *the reachability problem "$[h_1] \preceq_{\mathcal{F}} [h_2]$?" on $\mathcal{F}$ is decidable in time $\Phi(\max(|h_1|, |h_2|))$;*
2. *the similarity problem "$\langle [h'_1], [h'_2] \rangle \approx_{\mathcal{F}} \langle [h''_1], [h''_2] \rangle$?" is decidable in time $\Psi(\max(|h'_1|, |h'_2|, |h''_1|, |h''_2|))$;*
3. *for any terms h_1, h_2 the set $\{\langle [H_1], [H_2] \rangle / \approx_{\mathcal{F}} : [H_1 h_1] = [H_2 h_2]\}$ contains at most $\Theta(\max(|h_1|, |h_2|))$ similarity classes,*

where Φ, Ψ and Θ are monotonic recursive functions. Then the equivalence problem $\pi' \sim_{\mathcal{F}} \pi''$? is decidable in time $O((\Theta^2(n)\Phi(n^4\Theta(n))\Psi(n^4\Theta(n))n^6 \log n)$, where $n = \max(|\pi'|, |\pi'|)$.

Group frames. If $\preceq_{\mathcal{F}}$ is not a partial order on the set of states S of a semigroup frame $\mathcal{F}$, then the monoid $\mathcal{F}$ contains invertible elements. These elements form a subgroup in $\mathcal{F}$, and the decidability of the equivalence problem for PSPs on $\mathcal{F}$ depends essentially on this group.

Theorem 4 ([26,52]). *Suppose a homogeneous frame $\mathcal{F}$ is associated with a right-contracted monoid whose identity problem "$[g] = [h]$?" is decidable. Then the equivalence problem w.r.t. $\mathcal{F}$ is decidable iff it is decidable w.r.t. $\mathcal{F}'$, where $\mathcal{F}'$ is the maximal subgroup of $\mathcal{F}$.*

A uniform criterion for selecting groups with decidable problem of program equivalence was presented in [26,30].

Let $\mathcal{F}'\langle S, s_0, R\rangle$ be a frame corresponding to some group, $a_1, a_2, \ldots, a_k$ be a finite sequence of actions, and g, h be some terms. Then the sequence of states $[g], [ga_1], [ga_1, a_2], \ldots, [ga_1a_2 \ldots a_n]$ is called a *trajectory* in $\mathcal{F}'$. If $[ga_1a_2 \ldots a_n] = h$ then we say that this trajectory joins g to h. By the *distance* $\rho(g, h)$ between states g and h we shall mean the length of the shortest trajectory joining g to h. A set of states H, $H \subset S$, is called *complete* if for any h, $h \in H$, there exists a trajectory joining s_0 to h, all of whose elements belong to H, i.e. H is a simply connected subset of S containing the initial state s_0.

Let H be some complete set of states. Consider the relation $g <_H h$ on S which is true iff $g \neq h$ and each trajectory joining g to h passes through some state $[gh']$ such that $[h'] \in H$.

A frame $\mathcal{F}' = \langle S, s_0, R\rangle$ is called *β-frame* [30] if it contains a finite complete set H satisfying the following requirement:

> there exists a positive integer N such that, for any states g, h in $\mathcal{F}$ as soon as $g <_H h$ and $\rho(g, h) > N$ holds, there exists a state f which satisfies $g <_H f <_H h$.

Theorem 5 ([30]). *If $\mathcal{F}'$ is a β-frame then the equivalence problem for PSP on $\mathcal{F}'$ is decidable.*

It was demonstrated (see [17,26,30,52]) that a lot of well-known groups (finite groups, free groups, free commutative groups, direct products of free groups, etc.) fall into the category of β-groups and have a decidable problem of PSP equivalence. But in all these cases the complexity of the decision procedures is at least exponential of the size of programs.

On the other hand, Yu.Gurevich noticed that any Turing machine can be simulated by a PSP running on a non-trivial Abelian group. This provides a grounding for the following theorem.

Theorem 6 ([26]). *Suppose that a frame $\mathcal{F}$ is Abelian group and $rank(\mathcal{F}) \geq 2$. Then the equivalence problem for PSPs on $\mathcal{F}$ is undecidable.*

Letichevsky [26] showed that the equivalence problem for PSPs is decidable when $\mathcal{F}$ is Abelian group of rank 1.

Multitape automata. Combining some PSP's semantics that are easy for solving the equivalence problem one may get computational models which are extremely complicated for analysis. Thus, for example, by joining together a frame $\mathcal{F}_c$ based on a free commutative monoid and a valuation function ξ specified by Yanov shifts we get a model which has exactly the same computational power as deterministic multitape finite automata. In the framework of PSPs the semantics of multitape automata is specified by the following PDL axioms

$$\begin{aligned} [a_ia_j]Q &\equiv [a_ja_i]Q, \qquad a_i, a_j \in \mathcal{A} \\ p_j &\equiv [a_i]p_j, \qquad a_i \in \mathcal{A},\ p_j \in Sh(a_i) \end{aligned} \tag{2}$$

where a shift $Sh : \mathcal{A} \to 2^{\mathcal{P}}$ meets a requirement

$$a_i \neq a_j \;\Rightarrow\; Sh(a_i) \cap Sh(a_j) = \emptyset.$$

The notion of multitape finite automata was introduced by Rabin and Scott [45] in their classical paper of 1959. The equivalence problem for deterministic multitape finite automata is one of the most famous problem in automata theory. It remained open for more than 30 years. The difficulty of this problem is manifested in the following facts. In [45] it was noticed that the inclusion problem for deterministic multitape finite automata is undecidable. In 1968 Griffiths [20] proved the undecidability of the equivalence problem for nondeterministic multitape automata. On the other hand, by replacing axioms (2) by their weak variants

$$p_j \to [a_i]p_j, \qquad a_i \in \mathcal{A},\ p_j \in Sh(a_i),$$

one get a semantics composed of monotonic commutative models. In [42] it was demonstrated that the equivalence problem for PSPs on monotonic commutative models is decidable.

In [4,56] positive solutions to the two-tape case of the equivalence problem were given. It was demonstrated also (see [31,23,7]) that the equivalence problem is decidable on some specific classes of deterministic multitape finite automata. In 1991 Harju and Karhumaki reduced the equivalence problem for deterministic multitape automata to the multiplicity equivalence problem for one-tape automata, attacked the latter by generalizing Eilenberg' Equality Theorem [10], and finally proved the following theorem.

Theorem 7 ([13]). *The equivalence problem for the n-tape deterministic finite automata is decidable.*

To the extent of our knowledge there are no results so far on the complexity of the equivalence problem for multitape deterministic finite automata.

Reset actions. Suppose that $\mathcal{A}$ contains some distinguished actions $b_1, \ldots, b_k$ which reset any PSP's run to some prescribed states. These actions corresponds to the right zeros in monoids; their semantics is specified by the axioms

$$[a_j b_i]Q \equiv [b_i]Q, \qquad a_j \in \mathcal{A}.$$

The equivalence problem for semigroup frames with right zeros was studied in [25,16]. In [16] it was proved that the equivalence problem for PSPs with reset actions is decidable in time which is double-exponential of the size of programs to be analyzed. In [62] it was proved that this problem is decidable in time which is polynomial of the size of programs and exponential of the maximal number of occurrences of each reset action in programs. It was demonstrated also that a variety of problems in polynomial-time hierarchy (3-SAT, the emptiness problem for the intersections of n deterministic n-state finite automata) are interreducible with various cases of the equivalence problem for PSPs supplied with reset statements.

On the other hand, by providing deterministic two-tape finite automata with a single reset action we get one of the most simple deterministic model of computations whose equivalence problem is undecidable [36].

2 First-Order Sequential Programs

In this section we consider the concept of first-order sequential program (FOSP), its sintax, and semantics, and discuss the principal results on the equivalence problem for this model of computations.

2.1 Syntax of FOSP

Terms and substitutions. Fix three alphabets X, F, R of variables, function symbols, and relation symbols, respectively. Without loss of generality we assume that the set of variables $X = \{x_1, \ldots, x_N\}$ is finite.

The set of terms $Term_X$ over X and F is defined in the usual way. When a variable x_i occurs in a term t, we denote this fact by writing $x_i \in t$. A *substitution* on X is a mapping θ from X to $Term_X$. We write θ as a set of *bindings* $\{x_1/t_1, \ldots, x_N/t_N\}$, assuming, in contrast to [1], that $t_i = x_i$ is possible. The term t_i which binds a variable x_i in θ is denoted by $\theta[i]$. A set of variables $\{x_i : \ x_i \neq \theta[i]\}$ is called a *domain* $Dom(\theta)$ of a substitution θ. A set of variables $\{x_j : \ \exists k \ x_j \in \theta[j]\}$ is called a *range* of θ. We use ϵ to denote *empty substitution* $\{x_1/x_1, \ldots, x_N/x_N\}$, and write $\eta\theta$ for the composition of substitutions η and θ. The application of a substitution θ to an expression E (term, atom, formula) is defined as in [1] and denoted by $E\theta$. The set of all substitutions on X is denoted by $Subst_X^X$.

As far as sequential programs are concerned, a binding x_i/t_i may be thought of as an assignment $x_i := t_i$, while a substitution θ stands for a parallel composition of assignments.

Atoms and conditions. Formulae of the form $p(x_{i_1}, \ldots, x_{i_k})$, where p is a k-ary relation symbol and $x_{i_1}, \ldots, x_{i_k}$ are variables in X, are called *atoms*. A *literal* based on an atom A is either A itself or its negation $\neg A$. Consider some finite set of atoms $\mathcal{A} = \{A_1, \ldots, A_M\}$. Then a M-tuple $C = \langle L_1, \ldots, L_M \rangle$ of literals based on the atoms $A_1, \ldots, A_M$ is called a *condition*. We write $\mathcal{C}_\mathcal{A}$ for the set of all possible conditions based on the set $\mathcal{A}$ of atoms. Clearly, $|\mathcal{C}_\mathcal{A}| = 2^M$. Conditions stand for the conjunctions of basic tests used in conditional statements **if-then-else** and loop statements **while-do**.

Term interpretations. A *term interpretation* I for X, F, R is defined as follows (see [1]).

- the domain of I is the term universe $Term_X$;
- each k-ary function symbol f in F is assigned the mapping from $(Term_X)^k$ to $Term_X$ which maps a sequence $t_1, \ldots, t_k$ of terms to the term $f(t_1, \ldots, t_k)$;
- each k-ary relation symbol p in R is assigned a set p^I of k-tuples of terms.

Given a term interpretation I, an atom $A = p(x_{i_1}, \ldots, x_{i_k})$, and a substitution θ in $Subst_X^X$, we say that I *satisfies the atom* A (*the literal* $\neg A$) *for* θ iff $(x_{i_1}\theta, \ldots, x_{i_k}\theta) \in p^I$ (resp. $(x_{i_1}\theta, \ldots, x_{i_k}\theta) \notin p^I$) holds.

First-order sequential programs. Consider alphabets X, F, R, and a finite set of atoms $\mathcal{A}$. Then a *sequential program* over $X, F, \mathcal{A}$ is a finite-state labelled transition system represented by a tuple $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, \mathbf{loop}, S, T\rangle$, where

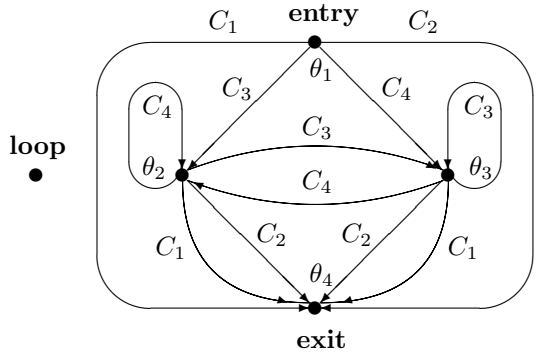
- V is a finite set of *program nodes*;
- $\mathbf{entry}, \mathbf{exit}, \mathbf{loop}$ are some distinguished program nodes;
- $S : V \to Subst_X^X$ is a total *assignment function* associating every program node with some substitution on X;
- $T : (V - \{\mathbf{exit}\}) \times \mathcal{C}_\mathcal{A} \to (V - \{\mathbf{entry}\})$ is a total *transition function* such that $T(\mathbf{loop}, C) = \mathbf{loop}$ holds for every condition C in $\mathcal{C}_\mathcal{A}$.

The nodes $\mathbf{entry}, \mathbf{exit}, \mathbf{loop}$ are called *initial, terminal*, and *dead* nodes, respectively. All other nodes in V are called *internal* nodes. By the *size* $|\pi|$ of a given program π we mean the total number of symbols involved in π. Given a finite sequence of program nodes $v_1, \ldots, v_{n-1}, v_n$ we write $S(v_1, \ldots, v_n)$ for the composition $S(v_n)S(v_{n-1}) \ldots S(v_1)$ of substitutions associated with the nodes.

Example 1. Consider a sequential program π written in traditional style.

```
i:=1; M:=0;
while i <= n
   do if a[i] > M then M:=a[i]; i:=i+1 od.
```

This program is intended for selecting the maximal positive element `M` in the integer array `a[1:n]`. According to the definition above it is represented by the following transition system over the set of variables $X = \{i, M\}$, the set of functions $F = \{a^{(1)}\}$, and the set of atoms $\mathcal{A} = \{A_1, A_2\}$, where $A_1 = (i \leq n)$ and $A_2 = (a(i) > M)$



$\theta_1 = \{i/1, M/0\}$,
$\theta_2 = \{i/i+1, M/M\}$,
$\theta_3 = \{i/i+1, M/a(i)\}$,
$\theta_4 = \epsilon$,
$C_1 = \langle \neg A_1, \neg A_2\rangle$,
$C_2 = \langle \neg A_1, A_2\rangle$,
$C_3 = \langle A_1, \neg A_2\rangle$,
$C_4 = \langle A_1, A_2\rangle$,

Fig. 1

2.2 Semantics of FOSP

Consider a program $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, \mathbf{loop}, S, T\rangle$. A sequence of pairs

$$w = (v_0, C_0), (v_1, C_1), \ldots, (v_i, C_i), \ldots, \tag{3}$$

where $v_0, v_1, \ldots, v_i, \ldots$ are program nodes and $C_0, C_1, \ldots, C_i, \ldots$ are conditions, is said to be a *path* in π if $v_0 = \mathbf{entry}$ and $v_{i+1} = T(v_i, C_i)$ holds for every $i,\ i \geq 0$. A path w is called *total* if either w is infinite or it is finite and has a pair $(\mathbf{exit}, C)$ as its last element.

Given a term interpretation I, we say that a total path (3) is a *run of* π *on* I if for every pair (v_i, C_i) the interpretation I satisfies each literal L in C_i for the substitution $S(v_0, v_1, \ldots, v_i)$. The pairs $(v_i, C_i),\ i \geq 0$, are said to be the *states* of the run. Whenever a state (v, C) occurs in (3) then we say that the run w *reaches* this state. Since the programs under consideration are deterministic, every program π has a unique run on a given interpretation I. Denote this run by $\pi(I)$. If $\pi(I)$ reaches the *final state* (v_m, C_m) such that $v_m = \mathbf{exit}$, then we say that the run *terminates* having the substitution $S(v_0, v_1, \ldots, v_m)$ as the *result*. Otherwise, when $\pi(I)$ is an infinite sequence, we say that it *loops* and has no result. We write $r(\pi, I)$ for the result of $\pi(I)$, assuming $r(\pi, I)$ is undefined when the run loops.

Programs π' and π'' are called *equivalent* ($\pi' \sim \pi''$ in symbols) if $r(\pi', I) = r(\pi'', I)$ for every term interpretation I. The *equivalence problem* for sequential programs w.r.t. term interpretations is to check for an arbitrary pair π_1, π_2 of programs whether $\pi' \sim \pi''$ holds.

Along with the equivalence relation some other properties of FOSP are important for computer program analysis. A program π is called *empty* (*total*) if $r(\pi, I)$ is undefined (respectively, defined) for every interpretation I. A program π is called *free* if every total path (3) in π is a run $\pi(I)$ of π on some interpretation I.

2.3 Decidable and Undecidable Cases of the Equivalence Problem for FOSPs

Undecidability results. An extensive study of the equivalence problem for a first-order model of sequential programs began since 1968, when M.Paterson [39] introduced a concept of formalized computer program and set up decision problems for this model of computations. The definition of FOSP above is but a modification of this concept expressed in terms of transition systems and substitutions. In 1970 D.C.Luckham, D.M.Park and M.S.Paterson [32] and Letichevsky [27] showed that multihead one-tape finite automata can be simulated by means of FOSP. Since the emptiness problem for multihead finite automata is undecidable, we arrive then at the following

Theorem 8 ([32,27]). *The equivalence, emptiness, totality and freedom problems for FOSP are undecidable.*

A serious effort was mounted to reveal to what extent program structure, alphabets and interpretations affect the undecidability of the main decision problems for FOSPs.

In [18,36,37,41] it was shown that the boarder between decidability and undecidability for FOSPs is very subtle.

When $|X| = 1$ and all functions and predicates are unary we have FOSPs with inseparable memory. Programs of this kind corresponds to Yanov schemata. When $|X| = n$, all predicates are unary, and all substitutions involved in programs are ones of the form $\{x_i/f(x_i)\}$, we obtain deterministic n-tape finite automata. The main decision problems for both classes of FOSP are, clearly, solvable. But as soon as some minor modifications are made in these computational models, there comes a point of undecidability.

Theorem 9 ([36]). *Let $X = \{x_1, x_2\}$, $F = \{f^{(1)}\}$, $P = \{p^{(1)}\}$. Suppose also that only substitutions of the form $\{x_1/f(x_1)\}$, $\{x_2/f(x_2)\}$, and $\{x_2/x_1\}$ are available in programs. Then the emptiness problem for FOSPs over (X, F, P) is undecidable.*

The alphabet $A_1 = (X, F, P)$ used in the theorem above is the minimal alphabet which guarantees the undecidability of the emptiness problem for FOSPs, since any reduction of A_1 leads to the well-known decidable cases of FOSPs. It should be noticed also that FOSPs over A_1 may be thought of as deterministic two-tape finite automata supplied with an extra action for inserting a content of one tape into another.

Theorem 10 ([37]). *Let $X = \{x_1, x_2\}$, $F = \{c^{(0)}, f^{(1)}, g^{(1)}\}$, $P = \{p^{(1)}\}$. Suppose also that only substitutions of the form $\{x_1/f(x_1)\}$, $\{x_2/g(x_2)\}$ and $\{x_1/c, x_2/c\}$ are available in programs. Then the totality problem for FOSPs over (X, F, P) is decidable, whereas the emptiness and the equivalence problems are not.*

In this theorem we deal with a class of FOSPs located on the very boarder between decidable and undecidable. Programs from this class correspond to deterministic two-tape finite automata supplied with a reset action.

Paterson's and Letichevsky's results [40,27] were greatly strengthened by Itkin and Zwinogrodski [21]. They showed that the equivalence problem for FOSPs is undecidable for any "reasonable" equivalence definition based on the set of all interpretations and applicable to any FOSP. It is assumed that an equivalence of FOSPs is "reasonable" if it meets the following requirements specified below informally.

1. An equivalence relation is said to *consistent* if non-equivalence of two programs π_1 and π_2 implies the existence of interpretation I for which runs $\pi_1(I)$ and $\pi_2(I)$ differ.
2. An equivalence relation is said to be *nonsingular* if the existence of statement which, in some interpretation I, would be essential for the result of the run $\pi_1(I)$ and non-essential for the result of the run $\pi_2(I)$ implies non-equivalence of π_1 and π_2.

Theorem 11 ([21]). *If the equivalence relation $\simeq$ on FOSPs is consistent and non-singular then the equivalence problem "$\pi_1 \simeq \pi_2$?" is undecidable.*

In the theory of first-order sequential programs this theorem plays exactly the same role as Rice's theorem for universal computer models.

Decidability results. The theorem above implies that decidable cases of the equivalence problem for FOSPs could be obtained either by turning from functional equivalence to some crude non-interpretive equivalences which are more readily solved, or by imposing additional constraints on the structure of programs and selecting thus some wide classes of programs with solvable problem of functional equivalence.

The first line of investigations has led to a concept of logic-term equivalence of FOSPs [22] which is as follows. For every path (3) going in a program $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, \mathbf{loop}, S, T\rangle$ from **entry** to **exit** a sequence

$$C_0S(v_0), C_1S(v_0, v_1), \ldots, C_iS(v_0, v_1, \ldots, v_i), \ldots$$

of conditions instantiated by compositions of substitutions $S(v_0, v_1, \ldots, v_i),\ i \geq 0$, assigned to program nodes $v_0, v_1, \ldots, v_i$ passed along this path, is referred to as a *determinant* of a path (3). Denote by $det(\pi)$ the set of determinants of all terminated paths in a program π. Two programs π_1 and π_2 are called *logic-term equivalent* if $det(\pi_1) = det(\pi_2)$.

It is apparent that $det(\pi)$ is a pure syntactical characteristic of a program π since it does not refer to any interpretations. It is easy to show also that $det(\pi_1) = det(\pi_2)$ implies $\pi_1 \sim \pi_2$. Therefore, the logic-term equivalence is a reasonable approximation to the interpretative equivalence. In [22] it was demonstrated that the logic-term equivalence can be checked effectively.

Theorem 12 ([22]). *The logic-term equivalence problem for FOSPs is decidable.*

A timed complexity of a decision procedure for the logic-term equivalence presented in [22] is double-exponential of the size of programs. Thereafter [5,29,49] the complexity of the logic-term equivalence checking procedures were improved substantially and reduced to $O(n^7)$. Due to the high efficiency of these algorithms, they are commonly used for data flow analysis and optimization of computer programs.

The alternative line of investigations provides the study of the equivalence problem for various classes of FOSPs. Results obtained in [27,32,36,37] show that the undecidability of the equivalence problem is induced by some specific features of substitutions involved in FOSP. By constraining the variety of substitutions used in programs we arrive at classes of FOSPs with decidable equivalence problem.

The first example of non-trivial class of programs with solvable equivalence problem was presented in [32,40]. A FOSP $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, \mathbf{loop}, S, T\rangle$ is called *progressive* if all substitutions θ involved in π are of the form $\{x/t\}$,

and for every pair of adjacent nodes u, v such that $S(u, C) = v$, an inclusion $Dom(T(u)) \subseteq Rang(T(v))$ holds.

Theorem 13 ([32,40]). *The equivalence problem for progressive programs is decidable.*

A FOSP π is called *conservative* if $x_i \in \theta[i]$ holds for every variable x_i, $x_i \in X$, and for every substitution θ involved in π. Conservative programs were introduced in [32]. Further investigations have shown that the decidability of the equivalence problem for FOSPs is closely related to conservative property.

In [28] it was demonstrated that the equivalence problem is decidable for conservative FOSPs whose atoms are of the same arity N, where $N = |X|$. In fact, this class of FOSPs corresponds to the computational model of PSPs whose semantics is based on partially commutative frames $\mathcal{F}_I$ (see [43]).

In [19] it was proved that the equivalence problem is decidable for conservative FOSPs whose atoms are of the form $p(x_1)$ and all bindings $\{x_1/t\}$ for the variable x_1 are such that $x_1 \in t$.

Since all decidable cases above deal with free FOSP, this has led M.Paterson to a belief that the equivalence problem should be decidable for free programs. Paterson's conjecture is not ruled out so far and gained new testimonies in its favor.

In [50] Sabelfeld introduced a class of *weakly conservative* programs which extends the class of conservative programs. A FOSP π is called weakly conservative if $Dom(\theta) \subseteq Rang(\theta)$ for every substitution θ involved in π.

Theorem 14 ([50]). *The equivalence problem for free weakly conservative programs is decidable.*

This theorem generalizes the results obtained in [28,19]. The only difficulty in its application is that the freedom problem for conservative programs is generally undecidable.

In [60] a class of orthogonal programs was introduced. We say that terms t and s *match* if one of the terms occurs in the other. Otherwise, a pair of terms t, s is said to be *orthogonal*. A term with no variables is called *ground*. We also say that a substitution θ is *orthogonal* if it is not a renaming, and each pair of terms $\theta[i], \theta[j]$, $1 \leq i < j \leq N$, is orthogonal.

Example 2. A substitution $\theta = \{g(x_2, a)/x_1, x_3/x_2, g(f(x_1, a), x_2)/x_3\}$ is orthogonal and a term $f(x_1, a)$ matches θ.

Given alphabets (X, F, R), $X = \{x_1, x_2, \ldots, x_N\}$, and a finite set of atoms $\mathcal{A}$ we say that a program $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, \mathbf{loop}, S, T\rangle$ over $X, F, \mathcal{A}$ is *orthogonal* if π meets the following requirements:

- every internal node v is associated with some orthogonal substitution $S(v)$ whose terms $S(v)[1], \ldots, S(v)[N]$ are non-ground;
- all atoms in $\mathcal{A}$ are ones of the form $p(x_1, \ldots, x_N)$, i.e. have all possible variables in X as their arguments.

An example of orthogonal program is depicted in Fig.1.

The class of orthogonal FOSP is in a sense dual to the class of progressive programs [32,40]. It is particularly remarkable that the orthogonal programs are free, though some of them are not weakly conservative.

Theorem 15 ([60]). *The equivalence problem for orthogonal programs is decidable.*

To the extent of our knowledge the classes of weakly conservative programs and orthogonal programs are the largest classes of FOSP with decidable equivalence problem.

3 Techniques for Proving Decidability/Undecidability of the Equivalence Problem

The undecidability results are usually obtained by applying common reduction technique to the well-known unsolvable problems (halting problem for Turing machines [26], emptiness problem for multihead finite automata [32], Post correspondence problem [21]).

A common way to prove the decidability of the equivalence problem for deterministic computational models is to find out a method for demonstrating the equivalence of two programs. This gives a semiprocedure for equivalence; the semiprocedure for nonequivalence trivially exists. The most-used techniques for proving the equivalence of sequential programs are "path" method, synchronization method, and reduction to a normal form.

Initially, the "path" method was introduced for proving lower bounds for computations [3]. This method takes an advantage of the specific phenomenon which is the very nature of some simple models of computation. It is as follows. Suppose that a program π (automaton) for some input data d has a run $r(\pi, d)$. If a length $|r(\pi, d)|$ exceeds some bound $L(\pi)$ which depends on program π only, then $r(\pi, d)$ can be broken down into fragments $r(\pi, d) = r_1 r_2 r_3$ so that joining the initial and the final fragments r_1 and r_3 we get a run of $r(\pi, d') = r_1 r_3$ of π for some other input data d'. This effect has much in common with Pumping Lemma which is widely used for proving non-regularity of formal languages [6]. It can be also applied for proving the equivalence of programs. Suppose that, given a pair of programs π_1, π_2, the results of some "long" runs of π_1 and π_2 reveal the difference between the functions realized by these programs. Then some "short" runs distinguish π_1 and π_2 as well. The equivalence problem is effectively solvable when the boundary between "long" and "short" computations depends recursively on some syntactic characteristics of π_1 and π_2. In this case one needs only to check the behaviour of programs on the finitely many runs to decide if they are equivalent or not. Usually the "path" method for equivalence checking is combined with the synchronization technique which reduces the equivalence problem to the emptiness problem for some devices that simulate synchronous runs of programs.

This approach was successfully employed in [16,5,22,25,28,30,40,45]. The main disadvantage of the "path" method is of its little use for practice since the number of "short" runs to be checked is very large (as a rule, it is exponential of the size of programs π_1 and π_2 under consideration). In [43,59,60,61] this deficiency has been overcome by encoding in algebraic terms the "difference" between intermediate states of computations $r(\pi_1, d)$ and $r(\pi_2, d)$. When a "difference" becomes too large, this indicates that results of $r(\pi_1, d)$ and $r(\pi_2, d)$ also differ. By choosing an appropriate encoding one could reduce the equivalence-checking problem "$\pi_1 \sim_{\mathcal{M}} \pi_2$?" to the well-known identity problem "$w_1 = w_2$?" in some specific monoids W associated with $\mathcal{M}$, and to the searching problem in a state space of polynomial size.

It is remarkable that whenever the "path" method gives a solution to the equivalence problem, then it always provides a solution to the inclusion problem, which is to find out whether π_1 and π_2 compute the same results whenever π_1 terminates. Since the inclusion problem for deterministic two-tape finite automata is undecidable, the "path" method is inapplicable to the equivalence problem for deterministic multitape finite automata.

The synchronization technique makes it possible to reduce the equivalence checking to the solution of the emptiness problem by constructing for given programs π_1 and π_2 an appropriate device $\pi_1 \times \pi_2$. It simulates synchronously computations of these programs so that a language accepted by $\pi_1 \times \pi_2$ is empty iff $\pi_1 \sim \pi_2$. The main difficulty in the using of this method is to provide the decidability of the emptiness problem for the crossproduct $\pi_1 \times \pi_2$. Nevertheless, in many cases (see [2,53,55,56,57,50,59,60]) this approach does work. A synchronization technique is discussed in much details in [8].

The key idea of a reduction to a normal form is in developing a complete equational calculus for an equivalence relation on programs. Usually, the completeness is proved by demonstrating that each pair of equivalent programs π_1 and π_2 can be transformed to the common normal form π_0. This approach has been used advantageously in [17,22,42,49,50,58].

References

1. K.Apt, From Logic Programming to Prolog, Prentice Hall, 1997.
2. E.Ashcroft, Z.Manna, A.Pnueli, A Decidable properties of monadic functional schemes, **J. ACM**, vol 20 (1973), N 3, p.489-499.
3. J.M.Barzdin, The complexity of recognition the symmetry by Turing machines. In *Problemy Kibernetiky*, **15**, 1965, p.245-248 (in Russian).
4. M.Bird, The equivalence problem for deterministic two-tape automata, *J. Comput. Syst. Sci.*, **7**, 1973, p.218-236.
5. A.O.Buda, Abstract computing machines. Tech. Report, Comp. Center of the USSR Academy of Science, Novosibirsk, N 108, 1978, 45 p (in Russian)..
6. D.Cozen, Automata and computability. Springer, 1997, 400 p.
7. K.Culik II, J. Karhumaki, HDT0L matching of computations of multitape automata, *Act. Inform.*, **27**, 1989, p.218-236.
8. K.Culik II, New techniques for proving the decidability of equivalence problems. *Theor. Comput. Sci.*, **71**, 1990, p.29-45.

9. J.W.De Bakker, D.A.Scott, A theory of programs. Unpublished notes, Vienna:IBM Seminar, 1969.
10. S.Eilenberg, Automata, Languages, and Machines. vol A, Academic Press, New-York, 1974.
11. A.P.Ershov, Theory of program schemata. In *Proc. of IFIP Congress 71*, Ljubljana, 1971, p.93-124.
12. D.Harel, Dynamic logics. In *Handbook of Philosophical Logics*, D.Gabbay and F.Guenthner (eds.), 1984, p.497-604.
13. T.Harju, J.Karhumaki, The equivalence of multi-tape finite automata. *Theoret. Comput. Sci.*, **78**, 1991, p.347-355.
14. Y.Hirshfeld, F.Moller, Decidable results in automata and process theory. *LNCS*, **1043**, 1996, p.102-148.
15. M.R.Garey, D.S.Johnson, A guide to the theory of NP-completeness, San-Francisco: W.H.Freeman & Company Publishers, 1979.
16. S.J.Garland, D.C.Luckham, Program schemes, recursion schemes and formal languages, *J. Comput. and Syst. Sci.*, **7**, 1973, p.119-160.
17. V.M.Glushkov, A.A.Letichevskii, Theory of algorithms and discrete processors, In *Advances in Information System Science*, vol 1 (1969), N 1.
18. A.B.Godlevsky, On some specific cases of halting problem and equivalence problem for automata. *Cybernetics*, 1973, N 4, p.90-97 (in Russian).
19. A.B.Godlevsky, On some special case of the functional equivalence problem for discrete transformers. *Cybernetics*, 1974, N 3, p.32-36 (in Russian).
20. T.V.Griffits, The unsolvability of the equivalence problem for λ-free nondeterministic generalized machines. *J. ACM*, **15**, 1968, p.409-413.
21. V.E.Itkin, Z.Zinogrodski, On program schemata equivalence. **J. Comput. and Syst. Sci.**, **6**, 1972, p.88-101.
22. V.E.Itkin, Logic-term equivalence of program schemata. *Cybernetics*, 1972, N 1, p.5-27 (in Russian)
23. E.Kinber, The inclusion problem for some classes of deterministic multitape automata. *Theoret. Comput. Sci.*, **26**, 1983, p.1-24.
24. A.A.Lapunov, Yu.I.Yanov, On logical program schemata, In *Proc. Conf. Perspectives of the Soviet Mathematical Machinery*, Moscow, March 12-17, 1956, Part III.
25. A.A.Letichevsky, On the equivalence of automata with final states on the free monoids having right zero. *Dokl. Akad. Nauk SSSR*, **182**, 1968, N 5 (in Russian).
26. A.A.Letichevsky, On the equivalence of automata over semigroup, *Theoretic Cybernetics*, **6**, 1970, p.3-71 (in Russian).
27. A.A.Letichevsky, Functional equivalence of finite transformers. II. *Cybernetics*, 1970, N 2, p.14-28 (in Russian)
28. A.A.Letichevsky, Functional equivalence of finite transformers. III. *Cybernetics*, 1972, N 1, p.1-4 (in Russian)
29. A.A.Letichevsky, Equivalence and optimization of programs. In *Programming theory*, Part 1, Novosibirsk, 1973, p. 166-180 (in Russian).
30. A.A.Letichevsky, L.B.Smikun, On a class of groups with solvable problem of automata equivalence, *Sov. Math. Dokl.*, **17**, 1976, N 2, p.341-344.
31. H.R.Lewis, A new decidable problem with applications. In *Proc 18th FOCS Conf.*, 1979, p.62-73.
32. D.C.Luckham, D.M.Park, M.S.Paterson, On formalized computer programs, *J. Comput. and Syst. Sci.*, **4**, 1970, N 3, p.220-249.
33. A.Mazurkiewicz, Trace theory. *LNCS*, **255**, 1987, p.279-324.

34. J.McCarthy, A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. Amsterdam:North-Holl. Publ. Co., 1963, p.33-70.
35. R.Milner, Processes: a mathematical model of computing agents. In *Proc. of Logic Colloquium'73*, 1973, p.157-174.
36. V.A.Nepomniaschy, On divergence problem for program schemes. *LNCS*, **45**, 1976, p.442-445.
37. V.A.Nepomniaschy, On the emptiness problem for program schemes. *Programming and Software Engineering*, 1977, N 4, p.3-13 (in Russian).
38. A.G.Oettinger, Automatic syntactic analysis and pushdown store. In Proc. Symposia on Applied Math., **12**, 1961.
39. M.S.Paterson, Programs schemata, *Machine Intelligence*, Edinburgh: Univ. Press, **3**, 1968, p.19-31.
40. M.S.Paterson, Decision problems in computational models, *SIGPLAN Notices*, **7**, 1972, p.74-82.
41. G.N.Petrosyan, On one basis of statements and predicates for which the emptiness problem is undecidable. *Cybernetics*, 1974, N 5, p.23-28 (in Russian).
42. R.I.Podlovchenko, On the decidability of the equivalence problem on a class of program schemata having monotonic and partially commutative statements. *Programming and Software Engineering*, 1990, N 5, p.3-12 (in Russian).
43. R.I.Podlovchenko, V.A.Zakharov, On the polynomial-time algorithm deciding the commutative equivalence of program schemata, *Reports of the Russian Academy of Science*, **362**, 1998, N 6 (in Russian).
44. V.R.Pratt, Semantical considerations of Floyd-Hoare logic. In *Proc. 17th IEEE Symp. Found. Comput. Sci.*, 1976, p.109-121
45. M.O.Rabin, D.Scott, Finite automata and their decision problems. *IBM J. Res. Dev.*, **3**, 1959, N 2, p.114-125.
46. H.G.Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, bf 89, 1953, p. 25-59.
47. H.Rogers, Theory of recursive functions and effective computability. McGraw-Hill, 1967.
48. J.D.Rutledge, On Ianov's program schemata, *J. ACM*, **11**, 1964, p.1-9.
49. V.K.Sabelfeld, Logic-term equivalence is checkable in polynomial time. *Reports of the Soviet Academy of Science*, **249**, 1979, N 4, p.793-796 (in Russian).
50. V.K.Sabelfeld, An algorithm deciding functional equivalence in a new class of program schemata, *Theoret. Comput. Sci.*, **71**, 1990, p.265-279.
51. G.Senizergues, The equivalence problem for deterministic pushdown automata is decidable, *LNCS*, **1256**, 1997, p.271-281.
52. M.A.Taiclin,The equivalence of automata w.r.t. commutative semigroups, *Algebra and Logic*, **8**, 1969, p.553-600 (in Russian).
53. E.Tomita, K.Seino, The extended equivalence problem for a class of non-real-time deterministic pushdown automata. *Acta Informatica*, **32**, 1995, p.395-413.
54. V.A.Uspensky, A.L.Semenov, What are the gains of the theory of algorithms: basic developments connected with the concept of algorithm and with its application in mathematics. *LNCS*, bf 122, 1981, p.100-234.
55. L.G.Valiant, Decision procedures for families of deterministic pushdown automata, Report N 7, Univ. of Warwick Computer Center, 1973.
56. L.G.Valiant, The equivalence problem for deterministic finite-turn pushdown automata, *Information and Control*, **25**, 1974, p.123-133.
57. L.G.Valiant, M.S.Paterson, Deterministic one-counter automata, *J. of Comput. and Syst. Sci.*, **10**, 1975, p.340-350.

58. J.Yanov, To the equivalence and transformations of program schemata, *Reports of the Soviet Academy of Science*, **113**, 1957, N 1, p.39-42 (in Russian).
59. V.A.Zakharov, The efficient and unified approach to the decidability of the equivalence of propositional programs. In *LNCS*, **1443**, 1998, p. 246-258.
60. V.A.Zakharov, On the decidability of the equivalence problem for orthogonal sequential programs, *Grammars*, **2**, 1999, p.271-281.
61. V.A.Zakharov, On the uniform technique for designing decision procedures for the equivalence of propositional sequential programs. In Proc. of the 4th Int. Conf. on Discrete Models in Control System Theory, Krasnovidovo, 2000, p.25-29 (in Russian).
62. V.A.Zakharov, On the equivalence problem for sequential program schemata supplied with reset statements. In Proc. of the 4th Int. Conf. on Discrete Models in Control System Theory, Krasnovidovo, 2000, p.153-154 (in Russian).

Two Normal Forms for Rewriting P Systems*

Claudio Zandron, Claudio Ferretti, and Giancarlo Mauri

DISCO - Universitá di Milano-Bicocca - Italy
zandron/ferretti/mauri@disco.unimib.it

Abstract. P systems have been recently introduced by Gh. Păun as a new model for molecular computations based on the notion of membrane structure. In this work, we consider Rewriting P Systems which use electrical charges to move objects between the membranes. We show that electrical charges, a feature introduced in order to obtain more "realistic" systems, induces simple systems: we define two normal forms for two variants of Rewriting P Systems which make use of electrical charges.

1 Introduction

The P systems were recently introduced in [1] 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.

Many variants are considered in [1], [5], [6], [7] and [8].

We consider here the systems introduced in [5] as Rewriting Super-Cell systems (or RP systems), in which the objects can be described by finite strings over a given finite alphabet, instead of objects of an atomic type (i.e. elements of a finite alphabet). The evolution of an object will correspond to a transformation of the string: the evolution rules are given as context-free rewriting rules. In a variant proposed in [6] the evolution rules do not specify the label of the

* This work has been supported by the Italian Ministry of University (MURST), under project "Unconventional Computational Models: Syntactic and Combinatorial Methods".

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 153–164, 2001.

membrane where the objects are sent. Instead, "electrical charges" are used, associated to membranes and to objects: they can be marked with "positive" (+), "negative" (−) or "neutral". An object marked with + (respectively −) will enter a membrane marked with − (respectively +), nondeterministically chosen from the set of membranes immediately inside to the membrane delimiting the region where the object is produced. The neutral objects are not introduced into an inner membrane.

As stated in [6], this feature is useful to obtain more "realistic" systems (with biochemical features instead than artificial ones). We show that this feature induces simple systems: we give two normal forms for RP systems using this feature. First, we will show that every language generated by a RP system with priority and without dissolving action can be generated by an equivalent system with two rules in each membrane and with a simple structure (the skin membrane contains elementary cells only). The second normal form is about RP systems without priority: we will show that every systems of that type (under certain restrictions) is equivalent to a system of the same type with three rules in each membrane.

A normal form for P-system can be found in [10].

2 Rewriting P Systems

A membrane structure is a construct consisting of several membranes placed in a unique membrane; this unique membrane is called a 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 by the membranes immediately inside it (the membranes are said to be *adjacent* to the delimited region). If we place multisets of objects in the regions 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. The *depth* of a P system is equal to the height of the tree describing its membrane structure.

We consider here *Rewriting Super-Cell Systems (RP Systems) with polarized membranes.* 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 string. Consequently, the evolution rules are given as rewriting rules. We only use context-free rewriting rules. The rules in a region can mark the objects to let them pass through the membranes as described in the following.

Such a system of degree n, $n \geq 1$, is a construct

$$\Pi = (V, \mu, M_1, \ldots, M_n, (R_1, \rho_1), \ldots, (R_n, \rho_n), i_0), \text{ where:}$$

- V is an alphabet
- μ is a membrane structure consisting of n membranes (labeled with $1, ..., n$); each membrane is marked with one of the symbols +, −, 0.

- $M_i, 1 \leq i \leq n$ are finite languages over V.
- $R_i, 1 \leq i \leq n$ are finite sets of context free evolution rules. They are of the form $X \to v(p)$ where X 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 $p \in \{here, out, +, -\}$. Often, the indication "here" is omitted. Note that the substring (p) is not inserted into the sentential form as subword. It is only used to communicate the objects through the membranes.
- $\rho_i, 1 \leq i \leq n$ are partial order relations over R_i
- i_0 is the output membrane

The membrane structure μ and the finite languages $M_1, ..., M_n$ constitute the *initial configuration* of the system. We can pass from a configuration to another one by using the evolution rules in parallel on all strings which can be rewritten, obeying the priority relations. Note that each string is processed by one rule only; the parallelism refers to processing simultaneously all available string by all applicable rules. If several rules can be applied to a string, 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.

A rule can mark 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 an *inner* membrane marked with $-$ (respectively $+$) and adjacent to the region where the rule is applied. If no such a membrane exists, then the rule cannot be applied.

If a rule contains the special symbol δ then the membrane where the rule is applied 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.

A sequence of transitions between configurations of a P system Π, is called a *computation* with respect to Π. A computation *halts* when there is no rule applicable to the objects present in the last configuration. The strings collected in the output membrane constitute the *output* of the P system. If a computation never stops, then it provides no output.

It is important to underline the fact that the evolution of strings is not independent, but interrelated in two ways:

1. If we have priorities, a rule r_1 applicable to a string x can forbid the use of another rule, r_2, for rewriting another string y which is present at the same time in the same membrane. Then, we can apply r_2 on y only if r_1 is not applicable to it and to the string x' obtained from x by using r_1.
2. Even without priority, if a string can be rewritten forever then all strings are lost, because the computation never stops.

We denote by $L(\Pi)$ the language generated by Π and by $RP^{\pm}(Pri, \delta)$ the family of languages generated by rewriting P systems using electrical charges, priority and the action indicated by δ. When one of the feature $\alpha \in \{Pri, \delta\}$ is not used,

we write $n\alpha$ instead of α. If only systems with at most k membranes are used, then we add the subscript k to RP.

More details and examples about P systems can be found in [4], [5] and [6]. For elements of Formal Language Theory, we refer to [11].

3 Rewriting P Systems with Priority and Polarized Membranes

In this section we show that every RE language can be generated by a RP system (with priority and without dissolving action) with exactly two rules in each region and of depth two.

In the following proof we use the notion of matrix grammar. 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 each sequence in M is called matrix). 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 [5], 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 [11]. Moreover, in [3] it is shown that the one-letter languages in MAT are regular.

Definition 1. *A RP system is in double_2_ normal_form if it is of depth 2 and in each membrane we have exactly 2 rewriting rules.*

Theorem 1. *Every RE language can be generated by a RP system (with electrical charges, with priority and without dissolving action) 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 $m_{k_2+1}, \ldots, m_k$ we label the matrices of type 4. Moreover, we label the symbols in N with $B_1, \ldots, B_h$.

We show how to construct a P system of depth 2 with exactly 2 rewriting rules in each membrane that generates the same language of G:

$$\Pi = (V, \mu, M_0, M_1, \ldots, M_k, M_{k+1}, \ldots, M_{k+h+1}, (R_0, \rho_0), (R_1, \rho_1), \ldots, (R_{k+h+1}, \rho_{k+h+1}), 0)$$

where

- $V = N_1 \cup N_2 \cup \{Z, Z', Z''\} \cup \{C_i | 1 \leq i \leq h\} \cup T$,
- $\mu = [_0[_1]_1^+ \ldots [_k]_k^+ [_{k+1}]_{k+1}^- \ldots [_{k+h+1}]_{k+h+1}^-]_0$
- $M_0 = \{Z'ZXA | S \to XA$ is the rule of a matrix of type 1$\}$
- $M_1, ..., M_{k+h+1}$ are empty
- $R_0 = \{r_{0,1} : Z \to \lambda(-)\} \cup \{r_{0,2} : Z' \to Z'(+)\}$
- $R_\alpha = \{r_{\alpha,1} : X \to ZY\} \cup \{r_{\alpha,2} : A \to x(out)\}$, $1 \leq \alpha \leq k_1$, ($m_\alpha : (X \to Y, A \to x)$ is a type 2 matrix)
- $R_\beta = \{r_{\beta,1} : A \to A\} \cup \{r_{\beta,2} : X \to ZY(out)\}$, $k_1 + 1 \leq \beta \leq k_2$, ($m_\beta : (X \to Y, A \to \dagger)$ is a type 3 matrix)
- $R_\gamma = \{r_{\gamma,1} : X \to C_1\} \cup \{r_{\gamma,2} : A \to x(out)\}$, $k_2 + 1 \leq \gamma \leq k$, ($m_\gamma : (X \to \lambda, A \to x)$ is a type 4 matrix)
- $R_\psi = \{r_{\psi,1} : B_{\psi-k} \to B_{\psi-k}\} \cup \{r_{\psi,2} : C_{\psi-k} \to C_{\psi-k+1}(out)\}, k + 1 \leq \psi \leq k + h - 1$,
- $R_{k+h} = \{r_{k+h,1} : B_h \to B_h\} \cup \{r_{k+h,2} : C_h \to Z''(out)\}$,
- $R_{k+h+1} = \{r_{k+h+1,1} : Z' \to \lambda\} \cup \{r_{k+h+1,2} : Z'' \to \lambda(out)\}$
- $\rho_i : r_{i,1} > r_{i,2}$, for every $0 \leq i \leq h + k + 1$

In other words, we place into the skin membrane several membranes with positive charge, one membrane 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 membrane for every nonterminal symbol in G, used to verify that the generated strings do not contain non terminal symbols. Finally we put one further membrane, with negative charge, to stop the computation.

Consider a string of the form $Z'ZHw$ 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 $Z \to \lambda(-)$

and we get the string $Z^{'}Hw$. This string is sent to a membrane with 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 $X \to ZY$ (which simulates the first production of the type 2 matrix) and a rule of the form $A \to x(out)$ (which simulates the second production of the type 2 matrix). The first production of a matrix of type 2 in the binary normal form cannot be of the form $X \to X$, as one can see from the description of the normal form in [2]. Thus, if the symbol X is in the string (i.e. $H = X$), 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 $X \to X$). Otherwise, we cannot apply this rule and the symbol Z is not reinserted in the string. Then, we have to apply the rule $A \to x(out)$. 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 $X \to ZY$ has not been applied, we have now a string of the form $Z^{'}Hw^{'}$, i.e. a string without the symbol Z. Thus, in the skin membrane we have to apply the rule $Z^{'} \to Z^{'}(+)$, which sends the string into a membrane with negative charge. It easy to see that there is no such a membrane which sends back the string in the skin membrane (the string does not contain 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 $X \to ZY$ and if the symbol A is in w, we can apply the rule $A \to x(out)$, 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 $A \to A$ and $X \to ZY$. We have the following possibilities: if the string contains the symbol A, we have to apply forever the production $A \to A$ (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 production $X \to ZY(out)$. If $H \neq X$ the string cannot further evolve and it will remain in this membrane forever, 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 $X \to C_1$ and $A \to x(out)$. 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 does not contain the symbol X, we can apply the rule $A \to x(out)$. The obtained string does not 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 $X \to C_1$ and then the rule $A \to x(out)$. 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 $Z \to \lambda$ cannot be applied, because the string does not contain the symbol Z. Thus, we have to apply the rule $Z^{'} \to Z^{'}(+)$. 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 B_1, that is the membrane $k+1$; it contains the productions $B_1 \to B_1$ and $C_1 \to C_2(out)$. 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 B_1 in the string: if B_1 is in the string, we have to apply forever the production $B_1 \to B_1$, otherwise we can apply the production $C_1 \to C_2(out)$ and we send back in membrane 0 the string $Z^{'}C_2w$. Here we can apply again the rule $Z^{'} \to Z^{'}(+)$ 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 B_2. If the string reaches another membrane, it will be blocked. If it reaches the membrane $k+2$ and the string contains the symbol B_2, the computation will never halt, otherwise we enter membrane 0 with the string $Z^{'}C_3w$. The computation proceeds in this way until we test all non terminal symbols. The sequence $C_1, C_2, \ldots, C_h$ permits us to be sure that we test the presence of all non terminal symbols.

In the membrane $k+h$ (which checks the presence of the $h-th$ non terminal symbol) 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 the 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 $L(G) = L(\Pi)$. □

As previously said, the system in the proof takes advantage of the polarization of the membranes and of the strings. The polarized strings are sent to membranes with an opposite charge (chosen in a nondeterministically way). This feature allows one to control the communication of the strings with only two general rules (in the skin membrane), because we do not have to specify the label of the membrane where the strings have to go (as in the models presented in [5]). With one rule we simulate a matrix and with the other we stop the simulation and we start the phase in which we control if the string is a terminal one. We can control if the string reaches a "wrong" membrane using the rules in the same membrane, as we have 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. It would be interesting to show how to obtain a system in the normal form in a direct way.

4 Rewriting P Systems without Priority

We consider now RP systems without priority and with *External Output*. This last feature were introduced in [9] and the differences with respect to the model previously described are the following:

- We do not collect the strings in an output membrane. Instead, we consider all the terminal strings which are sent out of the system at any time during the computation.
- If a string leaves the system but it is not terminal, then it is ignored; if a string remains in the system then it does not contribute to the language generated, even if it is a terminal one.
- We do not consider halting computations: we leave the process to continue forever and we observe the terminal strings which leaves the system; the language consists of all these strings.

We show that such a system can be reduced to a system with three rules in each membrane.

In the following we denote with $RPE^{\pm}(nPri, n\delta)$ the family of languages generated by Rewriting P systems with external output, which use electrical charges and which do not use priority relations nor dissolving membrane action.

Definition 2. *A P system is in 3_ normal_form if in each membrane we have exactly 3 rewriting rules.*

Theorem 2. *. Every language $L \in RPE^{\pm}(nPri, n\delta)$ can be generated by a RP system of the same type in 3_normal_form.*

Proof. Consider a RP system with external output

$$\Pi = (V, \mu, M_1, \ldots, M_k, R_1, \ldots, R_k)$$

We denote with m_i the $i-th$ membrane, and we assume that the skin membrane is m_1.

We show how to construct a RP system

$$\Pi^{'} = (V^{'}, \mu^{'}, M_1^{'}, \ldots, M_{k'}^{'}, R_1^{'}, \ldots, R_{k'}^{'})$$

($V^{'} = V \cup \{Z, Z_o, Z_o^{'}, Z_i, Z_+, Z_-\}$) in 3_normal_form, which generates the same language of Π.

To explain how we build Π', consider a membrane m_i of Π, with $i \neq 1$. Such a membrane contains a set of string M_i and a set of rules R_i. Moreover there are a number of cells placed immediately inside this one, some with positive charge and others with negative charge.

First of all, we replace every string w in m_i with a string Zw, where Z is a symbol not in V. We will use this symbol and the symbols $Z_o, Z_{o'}, Z_i, Z_+, Z_-$ to control the travel of the strings between the membranes. These symbols will be deleted before the string leaves the skin membrane, thus their position into the strings is of no interest (we need this later in the proof).

Then we place a membrane around the cells placed immediately inside m_i, in such a way that the rules in m_i remain outside this new membrane while the membranes in m_i will be inside this new membrane. We label this membrane with a_i (to indicate the membrane Around the membrane in i) and we give it a positive charge. In the new region just created, we place the rules:

– $Z_+ \to Z(+)$, $Z_- \to Z(-)$, $Z'_o \to Z(out)$

Then, we consider the rules in R_i. For every rule in R_i, we add a new membrane in the region defined by R_i and by the new membrane a_i, and we give negative charge to these membranes. If R_i contains q rules, we add q membranes. Each membrane will be used to simulate a rule. We denote these membranes with $mr_{i,j}$, where $1 \leq j \leq q$ (to indicate the Membrane added in i to simulate the Rule j). We have to simulate the rule and the travel of the obtained string too. For this reason, we get four different form of membrane (here, out, positive charge, negative charge).

For every rule of the form $A \to x(here)$ we add a membrane with the rules:

– $A \to Zx(out)$, $A \to ZA(out)$, $A \to A$

For every rule of the form $A \to x(out)$ we add a membrane with the rules :

– $A \to Z_o x(out)$, $A \to ZA(out)$, $A \to A$

For every rule of the form $A \to x(+)$ we add a membrane with the rules :

– $A \to Z_i Z_+ x(out)$, $A \to ZA(out)$, $A \to A$

For every rule of the form $A \to x(-)$ we add a membrane with the rules :

– $A \to Z_i Z_- x(out)$, $A \to ZA(out)$, $A \to A$

No string is initially present in these new membranes.

Note that in each membrane there is only one rule (the first one) that effectively simulates the corresponding rule in m_i. The two other rules are dummy rules introduced to get a system with exactly three rules in each membrane. As we will see later in the proof, they have no influence on the strings.

Once we have created these new membranes, we delete all the rule in m_i and we place in this membrane the rules:

– $Z \to \lambda(+)$, $Z_i \to \lambda(-)$, $Z_o \to Z'_o(out)$

The membrane obtained in this manner is denoted by $m_i^{'}$ (to indicate that the membrane has a corresponding membrane m_i in Π, and it is not a new one).

Thus, all membrane except the skin membrane are changed as described above. We modify the skin membrane in the same way with only one difference: the rule $Z_o \to Z_o^{'}(out)$ will not be added in $m_1^{'}$. Instead, we add the rule $Z_o \to \lambda(out)$.

To see how the computation proceeds, consider a string w in a membrane m_i. Using a rule in R_i we can replace a symbol in w to obtain a string w'. Then, w' can remain in the same membrane and it can be involved in a new rewriting rule or it can be sent to the membrane immediately outside or w' can be sent to a membrane immediately inside m_i (with positive or negative charge).

In the corresponding membrane of $\Pi^{'}$ (i.e. in $m_i^{'}$) we have the string Zw; on this string we can apply only the rule $Z \to \lambda(+)$; the obtained string w will be sent to a membrane $mr_{i,j}$, where we simulate one rewriting rule and we define where the obtained string have to be sent:

HERE If w reaches a membrane corresponding to a rule $A \to x(here)$ and w does not contain the symbol A, the string will remain in that membrane forever. Otherwise, we have the following possibilities:

- We can apply the rule $A \to Zx(out)$ to simulate the corresponding rule in m_i. The obtained string is of the form w_1Zw_2, where $w_1w_2 = w'$, and it will be sent outside to the membrane $m_i^{'}$, where we can apply again the rule $Z \to \lambda(+)$.
- We can apply the rule $A \to ZA(out)$. The obtained string is of the form w_3Zw_4, where $w_3w_4 = w$; it will be sent outside to the membrane $m_i^{'}$. The only difference between this string and the string Zw is the position of Z but, as said before, it is not important for the computation. Thus, this rule has no effect on the string.
- We can apply the rule $A \to A$. The string is not modified; it can be used many times with this rule, until we apply one of the other two rules, to send the string back in membrane $m_i^{'}$.

OUT If w reaches a membrane corresponding to a rule $A \to x(out)$ and w does not contain the symbol A, the string will remain in that membrane forever. Otherwise, we have the following possibilities:

- We can apply the rule $A \to Z_ox(out)$ to simulate the corresponding rule in m_i. The obtained string is of the form $w_1Z_ow_2$, where $w_1w_2 = w'$, and it will be sent back to membrane $m_i^{'}$. In membrane $m_i^{'}$ we have two possibilities:
  - If $i \neq 1$ (i.e. $m_i^{'}$ is not the skin membrane), we have to apply the rule $Z_o \to Z_o^{'}(out)$. We send the string $w_1Z_o^{'}w_2$ to the membrane immediately outside $m_i^{'}$, which is a membrane with a label a_v (v is the label of the membrane immediately outside the membrane i in Π). The membrane a_v is one of the membrane added in $\Pi^{'}$, originally not in Π. In membrane a_v, we can only apply the rule $Z_o^{'} \to Z(out)$, which sends the string w_1Zw_2 to the membrane outside, which is the membrane $m_v^{'}$. Thus, we correctly simulate the application of the rule and the obtained string

is sent to the correct membrane, where we can start the simulation of another rule.
  - If $i = 1$ (i.e. $m_i^{'}$ is the skin membrane), we have to apply the rule $Z_o \to \lambda(out)$. The symbol Z_o is deleted and the obtained string is sent outside the system; thus, if the string is a terminal one, it will be in the language $L(\Pi^{'})$.
- We can apply the rule $A \to ZA(out)$ or the rule $A \to A$. The consideration for the HERE case is still valid.

POSITIVE CHARGE If w reaches a membrane corresponding to a rule $A \to x(+)$ and w does not contain the symbol A, the string will remain in that membrane forever. Otherwise, we have the following possibilities:

- We can apply the rule $A \to Z_i Z_+ x(out)$ to simulate the corresponding rule in m_i. The obtained string is of the form $w_1 Z_i Z_+ w_2$, where $w_1 w_2 = w'$, and it will be sent outside to membrane $m_i^{'}$. Here we delete the symbol Z_i with the rule $Z_i \to \lambda(-)$. The string will be sent to the membrane with positive charge in $m_i^{'}$, i.e. the membrane a_i. In a_i we can apply only the rule $Z_+ \to Z(+)$. The obtained string $w_1 Z w_2$ will be sent to a membrane with negative charge, and the simulation of the rule $A \to x(+)$ is correctly done.
- We can apply the rule $A \to ZA(out)$ or the rule $A \to A$. The consideration for the HERE case is still valid.

NEGATIVE CHARGE This case is similar to the previous one.

Consider now the case that no rule in m_i (membrane of Π) can be applied to w. The string will remain blocked in m_i forever, thus no string will be generated from it. In the corresponding membrane $m_i^{'}$ (membrane of $\Pi^{'}$) we have the string Zw. We can apply the rule $Z \to \lambda(+)$ and the string w will be sent to a membrane $mr_{i,j}$. Because no rule in m_i can be applied to w, there is no membrane $mr_{i,j}$ with a rule that can be applied to it (the left symbol of the productions in $mr_{i,j}$ is the left symbol of a production in m_i), so the string w will remain blocked forever in the membrane.

Another possibility is that the computation on w in Π never halts, thus no string will be generated from w. In $\Pi^{'}$ we have two possibilities: the computation can continue forever by simulating the same rules or it can end, when the string w reaches a membrane $mr_{a,b}$ with no rules applicable on w. In both cases no string will be generated from Zw.

Thus, the strings that leave the skin membrane in $\Pi^{'}$ are the same of those in Π, that is $L(\Pi) = L(\Pi^{'})$. □

5 Conclusions

As pointed out in [1] and [4] the theory of computing with membranes misses basic tools (necessary conditions and normal form theorems) for producing examples and counterexamples.

We have made a little step in this direction by presenting in this paper two normal forms about Rewriting P-systems. The goal is to obtain simple systems

that are easy to analyze, in order to understand their generative power and their equivalence with other generative mechanisms. For instance, we have shown that every RP system of depth two and with two rules in each membrane can generate every RE language, by showing that such a system is equivalent to a Matrix Grammar System with Appearance Checking; moreover, we have shown that every RP system without priority is equivalent to a RP system with exactly three rules in each membrane. It is not known if RP systems without priority are able to generate RE language or not; we hope that the normal forms given here can be useful to give an answer to this question.

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. Gh Paun, Computing with membranes. An introduction, *Bulletin of the EATCS*, 67 (Febr. 1999), 139-152.
5. Gh. Paun, Computing with membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108-143, and *Turku Center for Computer Science-TUCS Report No 208*, 1998 (www.tucs.fi).
6. Gh. Paun, 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).
7. Gh. Paun, 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).
8. Gh. Paun, S. Yu, On synchronization in P systems, *Fundamenta Informaticae*, 38, 4 (1999), 397–410, and *University of Western Ontario Report TR 539*, 1999 (www.csd.uwo.ca/faculty/syu/TR539.html).
9. Gh. Paun, T. Yokomori, Membrane computing based on splicing, *proc. of 5th DIMACS Workshop on DNA Based Computers,* 1999, 213-227.
10. I. Petre, A normal form for P systems, *Bulletin of EATCS*, 67 (Febr. 1999), 165-172.
11. G. Rozenberg, A. Salomaa, eds. , Handbook of Formal Languages, *Springer-Verlag*, Heidelberg, 1997.

On a Conjecture of Kůrka. A Turing Machine with No Periodic Configurations*

Vincent D. Blondel[1], Julien Cassaigne[2], and Codrin Nichitiu[3]

[1] Division of Applied Mathematics, CESAME, Université catholique de Louvain,
4 avenue Georges Lemaître, B-1348 Louvain-la-Neuve, Belgium
blondel@inma.ucl.ac.be
[2] Institut de Mathématiques de Luminy — CNRS,
163 avenue de Luminy, Case 907, 13288 Marseille, France
cassaigne@iml.univ-mrs.fr
[3] EURISE, Université Jean Monnet St Étienne,
23, rue du Dr Paul Michelon, 42023 St Étienne Cedex 2, France
Codrin.Nichitiu@univ-st-etienne.fr

Abstract. A configuration of a Turing machine is given by a tape content together with a particular state of the machine. Petr Kůrka has conjectured that every Turing machine – when seen as a dynamical system on the space of its configurations – has at least one periodic orbit. In this paper, we provide an explicit counter-example to this conjecture. We also consider counter machines and prove that, in this case, the problem of determining if a given machine has a periodic orbit in configuration space is undecidable.

1 Introduction

A *Turing machine* is an abstract deterministic computer with a finite set Q of *internal states*. The machine operates on a doubly-infinite tape of cells indexed by an integer $i \in \mathrm{Z}$. A symbol taken from a finite alphabet Σ is written on every cell; a *tape content* can thus be seen as an element of Σ^{Z}.

At every discrete time step, the Turing machine scans the cell indexed by 0 and, depending upon its internal state and the scanned symbol, the machine either has no corresponding action, or performs one or more of the following operations: replace the scanned symbol with a new symbol, focus attention on an adjacent square by shifting the tape by one unit, and transfer to a new state. A Turing machine M can thus be given by its *transition function* $\delta_M : Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$. A tape content together with an internal state constitute a *configuration* of the machine. A Turing machine thus defines a (partial) function $f : Q \times \Sigma^{\mathrm{Z}} \to Q \times \Sigma^{\mathrm{Z}}$ on its configuration space $C = Q \times \Sigma^{\mathrm{Z}}$.

* This research was supported by NATO under grant CRG-961115, by the European Community Framework IV program through the research network ALAPEDES, by the Belgian program on interuniversity attraction poles IUAP P4-02, and by the French Ministry of Education and Research.

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 165–176, 2001.

This is actually one of several possible models of the Turing machine (TM), namely the TM with moving tape. Another model can be obtained by making the head scan a cell of any index (the tape being immobile), and by encoding the head position in the configuration. This is a TM with moving head, and the space of configurations has a different topology. We return to this issue in a few paragraphs.

We look at computing machines as dynamical systems on configuration space and look at the possible types of trajectories. This approach contrasts with the computability one, where the configurations are finite objects (the tape content is finite and the rest of the tape is filled with "blanks") and where we start the process from a precise initial configuration.

Let f be the function defined by some Turing machine M on its configuration space and let c be some configuration of M. The configuration $c' = f(c)$ is the *successor* of c. If f is not defined on c, then c has no successor and is said to be *terminal*. Denote by $f^t : C \to C$ the t-th iteration of f. A configuration c is *halting* (or *eventually terminal*) if $f^t(c)$ is terminal for some $t \geq 0$, it is *periodic* if $f^t(c) = c$ for some $t \geq 1$, and it is *eventually periodic* if $f^t(c)$ is periodic for some $t \geq 0$. Configurations that are not halting nor eventually periodic are said to be *wandering*. Thus, a configuration is either halting (1), eventually periodic (2), or wandering (3); see Figure 1.

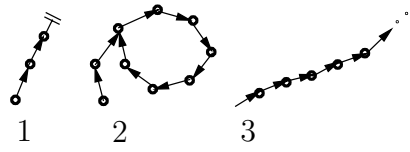


Fig. 1. The three possible types of configurations: 1. halting, 2. eventually periodic, and 3. wandering

It is clear that a machine has a periodic configuration if and only if it has an eventually periodic configuration, and this condition is again equivalent to that of the existence of a configuration that leads to periodic sequences of tape contents. Little is known about the possible combinations of configurations types defined by Turing machine. In [5], Petr Kůrka has proved that Turing machines (when endowed with a suitable topology) do not have attracting periodic orbits. In the same reference, Kůrka conjectures the following (see also [6] for a discussion of this conjecture):

Conjecture (Kůrka, 1997): A Turing machine that has no halting configuration has a periodic configuration.

In this paper, we analyze Kůrka's conjecture and questions related to it. For the Turing machine with moving head defined above there is a simple counter-example, the machine writing $0 \ldots 01$, with an increasing number of 0, which

answers the question for this model. From now on we study Turing machines with moving tape. We first consider counter machines rather than Turing machines. A n-counter machine with state set Q can be seen as a dynamical system on the configuration space $C = Q \times \mathrm{N}^n$. The 1-counter machine that keeps incrementing its unique counter has no periodic configuration and therefore constitutes an easy counter-example to a statement analogous to Kůrka's conjecture for counter machines. In Section 2, we prove the stronger result that, in the case of counter machines, the problem of determining if a given machine that has no halting configuration has a periodic configuration is undecidable.

In Section 3, we consider the Turing machine model. We provide an explicit construction of a Turing machine that has no halting configuration nor periodic configuration, thus disproving Kůrka's conjecture. The machine we construct has 36 states and operates on an alphabet of four letters.

2 Periodic Configurations for Counter Machines

A n-counter machine is an abstract deterministic computing machine with a finite set Q of internal states and a finite number of registers $R_1, \ldots, R_n$ containing nonnegative integers. The register values together with the internal state of the machine constitute a *configuration* of the machine. The configuration space of counter machines is thus given by $C = \mathrm{N}^n \times Q$. Depending upon its internal state and whether the registers are equal to 0, a machine can perform one of the following operations: leave the registers unchanged, increase some register R_j by 1, or decrease some register R_j by 1 (assuming $R_j \neq 0$), and move to a new internal state k. The transition function of the counter machine is defined through *instructions*. Instructions are tuples

$$[q, b_1, \ldots, b_n, j, D, k]$$

where $q \in Q$ represents the present state, $b_i \in \{true, false\}$ says whether the register R_i is equal to 0, j is the index of the register which is modified by the instruction, $D \in \{-1, 0, +1\}$ is the operation affecting R_j, and $k \in Q$ is the new internal state. For consistency, no two tuples begin with the same $n+1$ symbols. This definition of a counter machine is slightly different from that given in [4] but is easily seen equivalent in terms of computational power.

As explained in the Introduction, it is easy to construct counter machines that only have wandering configurations. Consider for example the 1-counter machine with one state, that keeps incrementing its unique register. This machine is defined by the two instructions

$$[1, true, 1, +1, 1]$$
$$[1, false, 1, +1, 1]$$

This machine has no halting nor periodic configuration; all its configurations are wandering. By adapting the proof of Theorem 1 in [2], we prove that distinguishing the counter machines that have a periodic configuration from those that have not such a configuration cannot be done algorithmically.

Theorem 1. *Let M be a counter machine that has no halting configuration. The problem of determining if M has a periodic configuration is undecidable. This problem is undecidable even in the case of 2-counter machines, but is decidable for 1-counter machines.*

Proof. The proof is by reduction from the classical halting problem for counter machines; see [4]. Consider a counter machine M with m internal states labeled $q_1, q_2, \ldots, q_m$, n registers $R_1, \ldots, R_n$ and let $s = (r_1, r_2, \ldots, r_n, q_l)$ be a given configuration of M. The instructions of M are, as said before, of the form $[q_i, b_1, b_2, \ldots, b_n, j, D, q_k]$.

To establish the first part of the result we describe how to effectively construct a counter machine M' that has no halting configuration, that has $n+2$ registers $R_1, \ldots, R_n, V, W$, and that has a periodic configuration if and only if M halts on s.

The machine M' has a special state denoted by q_0. Each time when it enters the state q_0, M' executes a sequence of instructions whose effect is to store r_i in R_i, for all i, $1 \leq i \leq n$, $2\max(1, V)$ in W and 0 in V. After having done this, the machine moves into state q_* and from there moves into state q_l. (The intermediate state q_* is only introduced to facilitate the exposition of the proof.)

Then the machine starts a simulation of the machine M. The simulation is such that, before performing any of the instructions of M, the machine first increases the value of the register V by 1, then tests the value of W. If the value of W is equal to 0 it returns to the special state q_0, otherwise it decreases this value by 1 and performs the instruction of the machine M. Thus, the instructions of the machine M, which are of the form $[q_i, b_1, b_2, \ldots, b_n, j, D, q_k]$, are all changed into twelve instructions for M':

$$[q_i, b_1, b_2, \ldots, b_n, b^*_{n+1}, b^*_{n+2}, n+1, +1, q'_i]$$
$$[q'_i, b_1, b_2, \ldots, b_n, b^*_{n+1}, true, n+2, 0, q_0]$$
$$[q'_i, b_1, b_2, \ldots, b_n, b^*_{n+1}, false, n+2, -1, q''_i]$$
$$[q''_i, b_1, b_2, \ldots, b_n, b^*_{n+1}, b^*_{n+2}, j, D, q_k]$$

where b^*_{n+1} and b^*_{n+2} range over all four possible combinations, that is $b^*_{n+1}, b^*_{n+2} \in \{true, false\}$. We complete the construction of M' by adding jumps to q_0 from all terminal configurations.

The machine M' we have constructed has no halting configuration. We claim that it has a periodic configuration if and only if M halts on s.

In order to prove our claim, assume first that M halts on s and let k be the number of steps after which it halts. Consider the machine M' at configuration $c = (r_1, r_2, \ldots, r_n, 0, 2k, q_*)$. Before every step simulation of M, the value of the register V is increased by 1 and that of W is decreased by 1. After k such steps, the values of the registers V and W are both equal to k and the machine M' jumps to q_0 with the value of V equal to k. From there it is easily verified that M' returns to c, and so c is a periodic configuration.

For the reverse implication, assume that M' has a periodic configuration. We first observe that all trajectories in configuration space pass infinitely many often through configurations of the form $(r_1, r_2, \ldots, r_n, 0, 2k, q_*)$ for some $k \geq 1$. Indeed, the register W is regularly decremented when executing instructions of M'. It is therefore clear that, whatever configuration the machine M' starts from, either the machine M' reaches a terminal configuration of M, or W reaches 0 after finitely many steps. In both cases, M' then jumps to q_0, executes a sequence of instructions whose effect is to store r_i in R_i, $2\max(1, V)$ in W, and 0 in V and finally moves to q_*; thus leading to the configuration $(r_1, r_2, \ldots, r_n, 0, 2k, q_*)$ for some $k \geq 1$.

From this observation we conclude that, if M' has a periodic configuration, then it must have one of the form $c = (r_1, r_2, \ldots, r_n, 0, 2k, q_*)$ for some $k \geq 1$. But a configuration of this type can only be periodic if the machine M halts on s after k steps. Indeed, if M does not halt on s, then the value of $2V + W$ regularly increases (it increases by 1 on each simulation step, and remains unchanged on other steps) and c is not periodic. Hence the result.

For 2-counter machines the proof uses the prime number encoding, and for 1-counter machines, the presence of a cycle is equivalent to the presence of a cycle where the value of the register is bounded by the number of states. □

3 A Counter-Example to Kůrka's Conjecture

We now describe our counter-example to Kůrka's conjecture. As a starting point, consider the Turing machine K_0 represented on Figure 2.

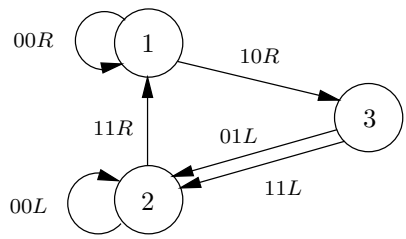


Fig. 2. The Turing machine K_0. The machine is defined by a labelled oriented graph. The vertices represent the states of the machine, and the labels on the arcs are used to define the transition function. For example, the label $10R$ on the arc from state 1 to state 3 means that, when in state 1 and reading the symbol 1, the machine writes the symbol 0 and shifts the tape one unit to the *left*, in order to read the symbol to the *right* of the 0 it has just written, then goes into state 3.

The machine has three states and operates on the two-letter alphabet $\{0, 1\}$. When in state 1, the machine searches for a 1 to the right of the head. Once it has found one, it changes it into a 0, writes a 1 to the right of it, and moves to state 3 from which it starts a left search for a 1. Once it has successfully

completed its search, it returns to state 1. It is easy to verify that this machine doesn't have any periodic configuration, except for the configurations for which the machine is in its left or right search state (state 1 or 3) and the tape is a tape of 0's. Machines that have search states do in fact always have periodic configurations whose associated tape is a periodic tape of symbols that do not satisfy the search. It is therefore clear that in constructing a counter-example we need to eliminate all search states. The machine we construct below essentially performs the same operations as those of K_0, except that it does so by bounding its searches. The technique we use for replacing searches by bounded searches is adapted from [3].

The Turing machine K_1 we construct has 36 states and operates on the four-letter alphabet $\{0, 1, 2, 3\}$. The machine is rather involved, and we show on Figure 3 only half of the machine; the other half is symmetrical, with L and R being interchanged. In order to describe the behavior of the machine, we have grouped the states in six groups denoted 1, 2, 3, $1'$, $2'$, $3'$. Each group i has a particular functional purpose, has a unique entry state, q_{i1}, and has two exit states: the failed search state q_{i5} and the dispatch state q_{i6}.

The group 1 has the same function as the state 1 of K_0 (search right for a non-zero symbol), the group 2 has the same function as the state 2 of K_0 (search left for a non-zero symbol), and the state q_{16} corresponds to state 3. In order to bound the searches without introducing periodic configurations we introduce a third group of states (group 3) that has no counterpart in K_0 and construct a symmetric set of states.

The technique for bounding the searches consist in using the searched zone on the tape as a stack. The stack is used to push the group index of failed searches. The corresponding pop operation is performed by the groups 3 (for right searches) and $3'$ (for left searches). The pop is a search operation as well, thus a push mechanism for it is also set in place. The states q_{ij} with $j = 1, 2, 3$ perform the bounded search. The states q_{ij} with $j = 4, 5$ write the 1 to be moved by the new search called from q_{i5}, and push the index of the failed search. The return from the new search to the calling search, when the one-unit move fails, is made from q_{16} (respectively $q_{1'6}$) by going to q_{31} (respectively $q_{3'1}$).

In the sequel, configurations of the machine will be given by expressions of the form $({}^{\omega}001\underline{0}100000^{\omega}, q_{11})$. This configuration for example is the one of tape content ${}^{\omega}0010100000^{\omega}$, internal state q_{11}, and with the head scanning the underlined symbol.[1] Note that, although we use here the usual notation for Turing machines with moving head, this is only for convenience and our model of Turing machines remains the moving tape model. Consequently, the underlined position always has index 0 and expressions such as $({}^{\omega}001\underline{0}100000^{\omega}, q_{11})$ and $({}^{\omega}00001\underline{0}1000^{\omega}, q_{11})$ represent the same configuration. As an illustration, the sequence of configurations obtained when starting from the state q_{11} on a tape of 0's is as follows:

[1] For $n \geq 0$, the notation 0^n is used to denote a sequence of n 0's. We use the notation 0^{ω} to denote an infinite sequence of 0's.

mark	tape content	group	state	purpose
	$^\omega 00\underline{0}0000000^\omega$	1	q_{11}	search (like state 1 of K_0)
	$^\omega 000\underline{0}000000^\omega$	1	q_{12}	search
	$^\omega 0000\underline{0}00000^\omega$	1	q_{13}	search and fail
	$^\omega 000\underline{0}100000^\omega$	1	q_{14}	prepare the new search
	$^\omega 00\underline{0}0100000^\omega$	1	q_{15}	push the index
$*$	$^\omega 001\underline{0}100000^\omega$	1	q_{11}	call a right search
	$^\omega 0010\underline{1}00000^\omega$	1	q_{12}	search and find
	$^\omega 00100\underline{0}0000^\omega$	1	q_{16}	erase and move (like state 3 of K_0)
	$^\omega 0010\underline{0}10000^\omega$	2	q_{21}	go back in order to bounce
	$^\omega 001\underline{0}010000^\omega$	2	q_{22}	search (like state 2 of K_0)
	$^\omega 00\underline{1}0010000^\omega$	2	q_{23}	search and find
	$^\omega 00\underline{1}0010000^\omega$	2	q_{26}	bounce and jump to right search
$*$	$^\omega 001\underline{0}010000^\omega$	1	q_{11}	search right again (like state 1 of K_0)
	$^\omega 0010\underline{0}10000^\omega$	1	q_{12}	search
	$^\omega 00100\underline{1}0000^\omega$	1	q_{13}	search and find
	$^\omega 0010000\underline{0}000^\omega$	1	q_{16}	erase and move
	$^\omega 00100\underline{0}1000^\omega$	2	q_{21}	go back
	$^\omega 0010\underline{0}01000^\omega$	2	q_{22}	search
	$^\omega 001\underline{0}001000^\omega$	2	q_{23}	search and fail
	$^\omega 0011\underline{0}01000^\omega$	2	q_{24}	prepare a new left search
	$^\omega 00110\underline{0}1000^\omega$	2	q_{25}	push the index
	$^\omega 0011\underline{0}21000^\omega$	$1'$	$q_{1'1}$	call a left search
	$^\omega 001\underline{1}021000^\omega$	$1'$	$q_{1'2}$	search and find
	$^\omega 00\underline{1}0021000^\omega$	$1'$	$q_{1'6}$	erase, but fail to move
	$^\omega 001\underline{0}021000^\omega$	$3'$	$q_{3'1}$	thus, pop the index
	$^\omega 0010\underline{0}21000^\omega$	$3'$	$q_{3'2}$	by searching for it
	$^\omega 00100\underline{2}1000^\omega$	$3'$	$q_{3'3}$	search and find
	$^\omega 00100\underline{2}1000^\omega$	$3'$	$q_{3'6}$	read and prepare to return
	$^\omega 0010\underline{0}01000^\omega$	2	q_{21}	pop, and return to left search group 2
	$^\omega 001\underline{0}001000^\omega$	2	q_{22}	resume the left search
	$^\omega 00\underline{1}0001000^\omega$	2	q_{23}	search and find
	$^\omega 00\underline{1}0001000^\omega$	2	q_{26}	bounce and jump to the right search
$*$	$^\omega 001\underline{0}001000^\omega$	1	q_{11}	restart the right search

In this example, we see in the $*$-marked lines that the machine passes through the configurations $(^\omega 01\underline{0}0^n 10^\omega, q_{11})$ for $n = 0, 1$ and 2. In the next section we prove that the machine will then pass through configurations of the type $(^\omega 01\underline{0}0^n 10^\omega, q_{11})$ for increasing values of n, so that the initial configuration $(^\omega 0\underline{0}0^\omega, q_{11})$ is not periodic; and more generally, whichever configuration it starts from, the machine passes through configurations where the head is at the beginning of increasingly larger blocks of zeros, and so the machine may not have periodic configurations.

We now prove that the machine K_1 doesn't have periodic configurations. Due to the symmetry of the machine, the proofs will only be detailed for one half of the configurations, the other being inferred by symmetry. We first need auxiliary definitions and results.

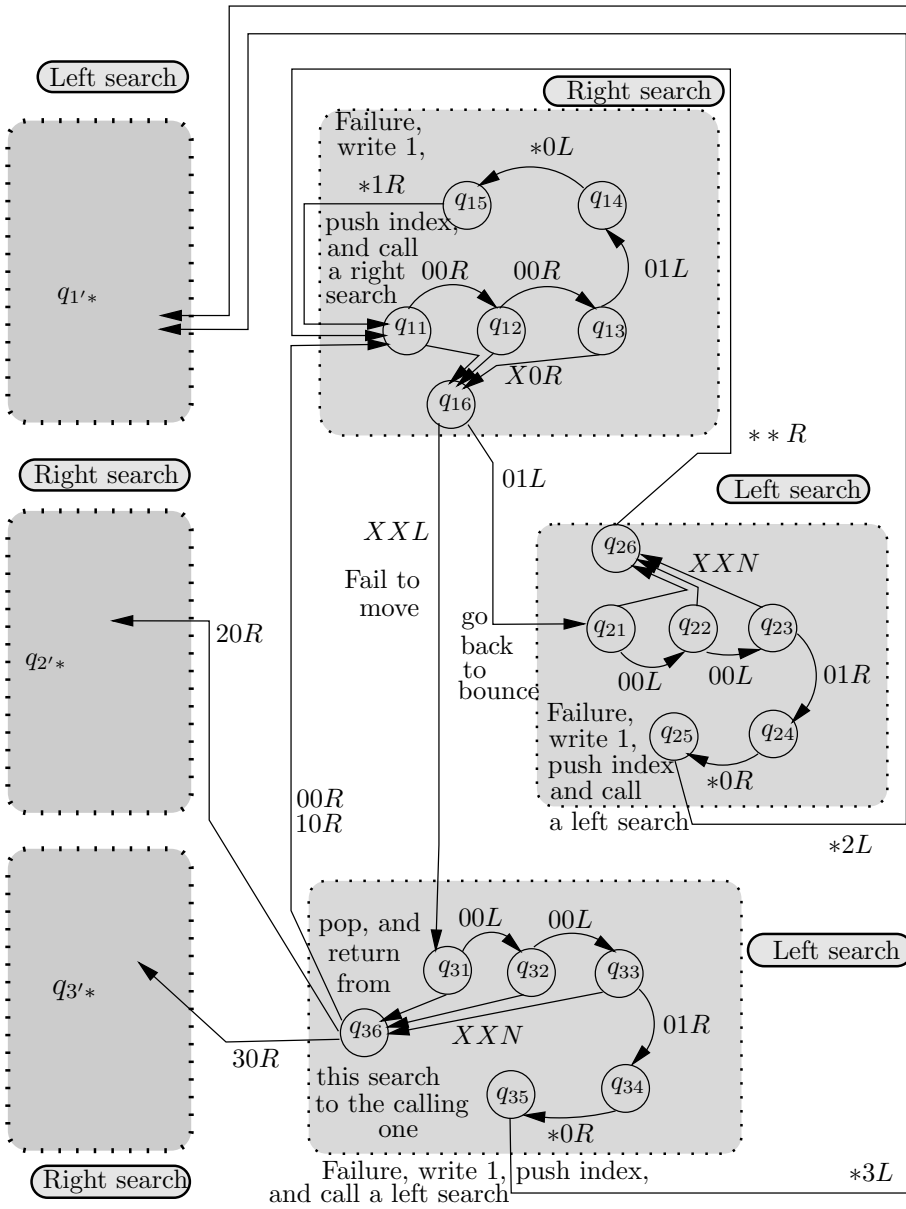


Fig. 3. The Turing machine K_1. Details are given only for half of the machine; the other half is symmetrical. L, R and N mean, respectively, right, left and no tape shift. We use the symbol $*$ to denote an arbitrary symbol of $\{0, 1, 2, 3\}$, and X to denote a symbol from $\{1, 2, 3\}$.

Definition 1. *For $s \geq 0$, let $Q(s)$ be the proposition:*

"The machine goes from any configuration of the form $(\ldots \underline{0}0^s XY \ldots, q_{11})$, with $X \in \{1,2,3\}$ and $Y \in \{0,1,2,3\}$, to configuration $(\ldots 0^{s+1}0\underline{Y} \ldots, q_{16})$, where the parts of the tape represented by ... remain unchanged. For $i = 2$ and $i = 3$, the machine goes from any configuration of the form $(\ldots X0^s\underline{0} \ldots, q_{i1})$ to configuration $(\ldots \underline{X}0^{s+1} \ldots, q_{i6})$. The symmetric statements also hold."

For $k \geq 2$, let $P(k)$ be the proposition:

"For any integers $t, p, n \geq 0$ such that $t + p + n + 2 = k$ and for any $X, Y, Z \in \{1,2,3\}$ the machine goes from configuration $(\ldots X0^t\underline{0}0^p Z0^n Y \ldots, q_{11})$ to configuration $(\ldots \underline{X}0^k Y \ldots, q_{36})$. The symmetric statement also hold."

Proposition 1. *Let $s \geq 0$. If $P(k)$ is true for all integers k with $2 \leq k \leq s$, then $Q(s)$ is true.*

Proof. The proof is by induction on s. The cases $s = 0$ and $s = 1$ can be checked by hand by following the machine's diagram on Figure 3.

Then, to prove the proposition by induction on s it is sufficient to show that ($Q(s-1)$ and $P(s)$) implies $Q(s)$. We have the following configuration sequence:

$$
\begin{array}{ll}
\ldots \underline{0}000^{s-2}XY \ldots & q_{11} \\
\quad\downarrow & \text{A few iterations} \\
\ldots 1\underbrace{\underline{0}10^{s-2}}_{s}XY \ldots & q_{11} \\
\quad\downarrow & P(s) \text{ (hypothesis)} \\
\ldots \underline{1}000^{s-2}XY \ldots & q_{36} \\
\quad\downarrow & \text{One iteration} \\
\ldots 0\underline{0}00^{s-2}XY \ldots & q_{11} \\
\quad\downarrow & Q(s-1) \text{ (hypothesis)} \\
\ldots 0000^{s-2}0\underline{Y} \ldots & q_{36}
\end{array}
$$

The proof for the configurations $(\ldots \underline{0}0^s X \ldots, q_{21})$ and $(\ldots \underline{0}0^s X \ldots, q_{31})$ is similar. □

Lemma 1. *$P(k)$ is true for all $k \geq 2$.*

Proof. Define $\widehat{P}(k,n)$ to be the proposition $P(k)$ where n is also fixed. Let $L = \{(k,n) \,|\, n \geq 0, n+2 \leq k\}$ and consider the lexicographical order $<_L$, i.e., $(k',n') <_L (k,n)$ when $k' < k$, or when $k' = k$ and $n' < n$. The order $<_L$ is well-founded, therefore allowing us to prove the lemma by induction.

Fix (k,n) from L and assume $\widehat{P}(k',n')$ true for all $(k',n') \in L$ such that $(k',n') <_L (k,n)$. Note that this implies, in particular, that $P(k')$ is true for all $k' < k$ and so, by Proposition 2, $Q(s)$ is true for all $s \leq k-1$.

Case $n = 0$ By using $Q(p)$, one iteration and then $Q(k-1)$, we get from $(\ldots X0^{k-p-2}\underline{0}0^p ZY \ldots, q_{11})$ to $(\ldots \underline{X}0^{k-p-2}00^p 0Y \ldots, q_{36})$.

Case $n > 0$ Again, by using $Q(p)$, one iteration, $Q(k-1-n)$, one iteration, and then $\widehat{P}(k, n-1)$, we get from $(\ldots X0^{k-n-p-2}\underline{0}0^pZ00^{n-1}Y\ldots, q_{11})$ to $(\ldots \underline{X}00^{k-n-p-2}0^p000^{n-1}Y\ldots, q_{36})$. In both cases $\widehat{P}(k, n)$ holds. This concludes the proof. □

Putting together Proposition 2 and Lemma 1 we immediately conclude:

Lemma 2. *$Q(s)$ is true for all $s \geq 0$.*

We now prove that the machine doesn't have periodic configurations of a particular type.

Lemma 3. *The configurations $(\ldots \underline{0}0^\omega, q_{ij})$ with $i \in \{1, 2', 3'\}$ and $j \in \{1, 2, 3\}$ are not periodic. This is also true for the symmetric case.*

Proof. Starting from the configuration $(\ldots \underline{0}0^\omega, q_{ij})$ for some $i \in \{1, 2', 3'\}$ and $j \in \{1, 2, 3\}$, the machine goes to configuration $(\ldots X\underline{0}10^\omega, q_{11})$ where, depending on i, $X = 1, 2$ or 3. This configuration is of the type $(\ldots X\underline{0}0^pY0^\omega, q_{11})$ for some $X, Y \in \{1, 2, 3\}$. Using $Q(p)$, one iteration, $Q(p+1)$, and another iteration, we get from $(\ldots X\underline{0}0^pY00^\omega, q_{11})$ to $(\ldots X\underline{0}0^p010^\omega, q_{11})$. By iteratively applying this result, we extract from the configuration sequence obtained a subsequence of configurations of the type $(\ldots X\underline{0}0^p10^\omega, q_{11})$ with strictly increasing values of p. Therefore the sequence is not periodic. □

In order to prove our main theorem, we need one more lemma.

Lemma 4. *Started in configuration $(\ldots \underline{0}0^{n-1}\ldots, q_{i1})$ or $(\ldots 0^{n-1}\underline{0}\ldots, q_{i'1})$, with $n \geq 1$, $i \in \{1, 2', 3'\}$ and $i' \in \{1', 2, 3\}$, the machine goes to a configuration $(\ldots \underline{0}0^n\ldots, q_{j1})$ or $(\ldots 0^n\underline{0}\ldots, q_{j'1})$, for some $j \in \{1, 2', 3'\}$ or $j' \in \{1', 2, 3\}$. Here, contrarily to definition 1, the parts of the tape represented by ... can be modified in the process.*

Proof. We only consider the cases $(\ldots \underline{0}0^{n-1}\ldots, q_{i1})$ for $i = 1, 2', 3'$. Let X be the symbol on the tape to the right of $\underline{0}0^{n-1}$. If $X = 0$, we are already in a configuration $(\ldots \underline{0}0^n\ldots, q_{j1})$. Let us now assume that $X \in \{1, 2, 3\}$. There are three cases to consider. Let Z and T denote arbitrary symbols.

Case $i = 1$	$\ldots \underline{0}0^{n-1}XZ\ldots$	q_{11}	line 1A
	$\downarrow$		$Q(n-1)$
	$\ldots 00^{n-1}0\underline{Z}\ldots$	q_{16}	
	$\downarrow$		one iteration
	$\ldots 00^{n-1}\underline{0}1\ldots$	q_{21}	if $Z = 0$
or	$\ldots 00^{n-1}\underline{0}Z\ldots$	q_{31}	if $Z \in \{1, 2, 3\}$

Case $i = 2'$	$\ldots ZT\underline{0}0^{n-2}0X\ldots$	$q_{2'1}$	line 1B
	$\downarrow$		$Q(n-1)$
	$\ldots ZT00^{n-2}0\underline{X}\ldots$	$q_{2'6}$	
	$\downarrow$		one iteration
	$\ldots ZT00^{n-2}\underline{0}X\ldots$	$q_{1'1}$	

If $T \neq 0$, this is exactly symmetrical to line 1A.If $T = 0$, we are in the desired configuration $(\dots 0^n \underline{0} \dots, q_{j'1})$.

$$\begin{array}{lll} \text{Case } i = 3' & \dots ZT\underline{0}0^{n-2}0X\dots & q_{3'1} \\ & \downarrow & Q(n-1) \\ & \dots ZT00^{n-2}0\underline{X}\dots & q_{3'6} \\ & \downarrow & \text{one iteration} \\ & \dots ZT00^{n-2}\underline{0}0\dots & q_{k1} \end{array}$$

If $T = 0$ we are done, so let us assume $T \in \{1, 2, 3\}$. The value of k depends on X. If $X = 1$ (respectively, $X = 2$), then $k = 1'$ (respectively, $k = 2$) and we conclude with the symmetric of line 1A (respectively, line 1B). If $X = 3$, then $k = 3$, and the sequence continues (using $Q(n-1)$) to $(\dots Z0\underline{0}0^{n-2}00\dots, q_{j1})$ for some j (depending on T), which concludes the proof of the lemma. □

We can now finally prove:

Theorem 2. *The machine K_1 has no periodic configuration.*

Proof. Consider an arbitrary configuration c. After at most 4 steps, the machine enters a state q_{i1} for some $i \in \{1, 2, 3, 1', 2', 3'\}$. A brief case analysis shows that if the scanned symbol is non-zero, after at most 4 other steps, no matter the symbols around the scanned one, the machine returns in a state q_{i1} with a zero as scanned symbol. Then we are in the situation described in Lemma 4, and for all n the machine will reach a configuration of the form $(\dots \underline{0}0^{n-1}\dots, q_{i1})$ with $i \in \{1, 2', 3'\}$, up to symmetry. Either at some point the machine will reach a configuration of the form $(\dots \underline{0}0^{\omega}, q_{i1})$, up to symmetry, which by Lemma 3 is not periodic; or we can extract an infinite sequence of configurations $(\dots \underline{0}0^{n-1}X\dots, q_{i1})$, up to symmetry, with $X \neq 0$ and increasing n, so that the initial configuration is not periodic. □

Let us notice that while the machine K_1 has 36 states and operates on a four-letter alphabet, the construction can be changed so that it uses either only two letters with more states, or only three states with more letters (by using a result from [8]).

4 Discussion

Various models of computing devices (Turing machines, counter machines, cellular automata, recurrent artificial neural networks, etc.) are equivalent in terms of computational power when they are seen as defining functions from input to output. In this paper, we consider the *dynamics* of some computing devices. We do not just look at the relations between inputs and outputs, but also look at what happens in between. We also look at configurations that do not correspond to a valid input (e.g., infinite tape content). The answer to the question we consider in this context (the existence of periodic configurations) highly depends on the chosen computing model. For example, recurrent neural networks always

have a periodic configuration (a periodic configuration is given by the fixed point for which all activation levels are set equal to 0). For these networks one can also infer from Theorem 1 of [1] the existence of networks that have no other periodic configuration. The situation for cellular automata is quite different. For a given cellular automaton, there are only finitely many space-periodic configurations of a given period. Space-periodicity is preserved through iteration, and so all space-periodic configurations of cellular automata are periodic and cellular automata always have infinitely many periodic configurations. In the case of Turing machine with moving tape, we have shown that, contrarily to what was conjectured, not all Turing machines have a periodic configuration. Finally, in the case of counter machines we have proved that identifying the presence of a periodic configuration is an undecidable task. These few examples show how rich and different the dynamics of these computing devices are.

Acknowledgment

The authors are grateful to Professor Maurice Margenstern for his suggestions of improvement.

References

1. V. D. Blondel, O. Bournez, P. Koiran, and J. N. Tsitsiklis. The Stability of Saturated Linear Dynamical Systems is Undecidable. Proceedings of STACS 2000, H. Reichel, S. Tison (Eds), Lecture Notes in Computer Science, Springer Verlag, Heidelberg, **1770** (2000), 479–490. Full version to appear in *Journal of Computer and System Sciences*.
2. V. D. Blondel, O. Bournez, P. Koiran, Ch. H. Papadimitriou, and J. N. Tsitsiklis. Deciding Stability and Mortality of Piecewise Affine Dynamical Systems. To appear in *Theoretical Computer Science*.
3. P. K. Hooper. The Undecidability of the Turing Machine Immortality Problem. *The Journal of Symbolic Logic*, **31**(2) (1966), 219–234.
4. J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*, Addison-Wesley, 1969.
5. P. Kůrka. On Topological Dynamics of Turing Machines. *Theoretical Computer Science*, **174** (1997), 203–216.
6. C. Moore. Finite-Dimensional Analog Computers: Flows, Maps, and Recurrent Neural Networks, in *Proc. of the First International Conference on Unconventional Models of Computation*, Auckland, New Zealand, 1998.
7. C. Moore. Generalized Shifts: Undecidability and Unpredictability in Dynamical Systems. *Nonlinearity*, **4** (1991), 199–230.
8. Y. Rogozhin. Undecidability of the immortality problem for Turing machines with three states. *Cybernetics (Kibernetika)*, Kiev (USSR) **1** (1975), 41–43.

On the Transition Graphs of Turing Machines

Didier Caucal

IRISA–CNRS, Campus de Beaulieu, 35042 Rennes, France
caucal@irisa.fr

Abstract. As for pushdown automata, we consider labelled Turing machines with ϵ-rules. With any Turing machine M and with a rational set C of configurations, we associate the restriction to C of the ε-closure of the transition set of M. We get the same family of graphs by using the labelled word rewriting systems. We show that this family is the set of graphs obtained from the binary tree by applying an inverse mapping into F followed by a rational restriction, where F is any family of recursively enumerable languages containing the rational closure of all linear languages. We show also that this family is obtained from the rational graphs by inverse rational mappings.

1 Introduction

The transition graphs of some classes of machines have already been investigated. First, Muller and Schupp have considered rational restrictions of the transition graphs of pushdown automata: these graphs are the graphs of bounded degree having a finite number of non isomorphic connected components when decomposed by distance from any vertex ; these graphs have a decidable monadic theory [15]. This graph family is also the set of rational restrictions of the prefix transition graphs of finite labelled word rewriting systems [7]. Extending to recognizable labelled rewriting systems, the rational restrictions of their prefix transition graphs define a larger family of graphs having a decidable monadic theory [8]. To extend this last family, Morvan has defined the family of rational graphs, which are the graphs recognized by transducers with labelled outputs [14]. This family is general: it contains for instance the transition graphs of Petri nets, the transition graphs of congruential systems [17], and the transition graphs of labelled word rewriting systems. Although the rational graphs are recursive, they have in general an undecidable first order theory.

There is a simple and uniform way to present all the previous families of graphs from families of languages. Take a family F of languages, a set T of labels and a mapping $h : T \longrightarrow F$ associating to each label a language in F. The inverse image $h^{-1}(G)$ of a graph G by h has an arc $s \xrightarrow{a} t$ when there is a path $s \xRightarrow{u} t$ in G for some word u in $h(a)$. A family F of languages induces a family REC_F of graphs obtained from the infinite binary tree by marking rationally some vertices (with a special letter) and then applying an inverse mapping. Then the family of rational restrictions of the transition graphs of pushdown automata

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 177–189, 2001.

is the family REC_{Fin} of graphs induced by the family Fin of finite languages. Furthermore, the family of rational restrictions of the prefix transition graphs of recognizable rewriting systems is the family REC_{Rat} of graphs induced by the family Rat of rational languages [8]. Finally, the family of rational graphs is the family $REC_{\overline{Lin}}$ of graphs induced by a subfamily $\overline{Lin}$ of linear languages [14].

Thus small families of languages induce large families of graphs. Conversely, a family of graphs yields a family of languages. A trace of a graph is the language of path labels from and to given finite vertex sets. The traces of finite graphs are the rational languages. The traces of graphs in REC_{Fin} are the context-free languages which are also the traces of graphs in REC_{Rat}. Finally, the traces of rational graphs are context-sensitive languages but the converse is a conjecture: is any context-sensitive language the trace of a rational graph ?

Following the Chomsky hierarchy, we present a general family of graphs, the traces of which are the recursively enumerable languages. We consider the off-line Turing machines investigated by Mateescu and Salomaa [13] with a read only one way input tape and a unique two ways working tape. These machines are particular labelled word rewriting systems allowing rules labelled by ε. Following the work of Payet [16] and as for prefix transition graphs of word rewriting systems, we consider rational restrictions of the ε-closure of transition graphs of these off-line Turing machines. We show that this family of graphs coincides with the family of rational restrictions of the ε-closure of the transition graphs of word rewriting systems. We also show that this graph family is equal to $REC_{Rat(\overline{Lin})} = REC_{RE}$ meaning that it is induced by any language family between the rational closure of $\overline{Lin}$ and the family RE of recursively enumerable languages. Finally, we show that this graph family is obtained by inverse rational mappings of rational graphs.

2 Preliminaries on Graphs

Let P be a subset of a monoid M, and $Id_P = \{ (u,u) \mid u \in P \}$ the *identity* relation on P. A (simple oriented labelled) *P-graph* G is a subset of $V \times P \times V$ where V is an arbitrary set. Any (s,a,t) of G is a *labelled arc* of *source* s, of *target* t, with *label* a, and is identified with the labelled transition $s \xrightarrow[G]{a} t$ or directly $s \xrightarrow{a} t$ if G is understood. We denote by $V_G := \{ s \mid \exists\, a\, \exists\, t,\ s \xrightarrow{a} t \vee t \xrightarrow{a} s \}$ the *vertex* set of G. The set $2^{V \times P^* \times V}$ of P^*-graphs with vertices in V is a monoid for the *composition* $G \circ H := \{ r \xrightarrow{a \cdot b} t \mid \exists\, s,\ r \xrightarrow[G]{a} s \wedge s \xrightarrow[H]{b} t \}$ for any $G, H \subseteq V \times P^* \times V$, where $\{ s \xrightarrow{1} s \mid s \in V \}$ is its neutral element. The submonoid $\{G\}^*$ of $2^{V_G \times P^* \times V_G}$ generated by any graph G gives by union the graph $G^* := \bigcup \{G\}^*$. The relation $\xrightarrow[G^*]{u}$ denoted by $\overset{u}{\underset{G}{\Longrightarrow}}$ or simply by $\overset{u}{\Longrightarrow}$ if G is understood, is the existence of a *path* in G labelled $u \in P^*$. For every $Q \subseteq P^*$, we write $s \overset{Q}{\Longrightarrow} t$ if there is some $u \in Q$ such that $s \overset{u}{\Longrightarrow} t$. The *restriction* $G_{|C}$ of a P-graph G to an arbitrary set C is $G_{|C} := G \cap (C \times P \times C)$. The labels $L(G,E,F)$

of paths of G from a set E to a set F is the set $L(G,E,F) := \{\ u \in M \mid \exists\ s \in E,\ \exists\ t \in F,\ s \underset{G}{\overset{u}{\Longrightarrow}} t\ \}$. A *trace* of a graph G is the language $L(G,E,F)$ of path labels from a finite set E to a finite set F.

Given an alphabet T and a relation $h \subseteq T \times P$, the inverse $h^{-1}(G)$ by h of any P-graph G is the following T-graph:

$$h^{-1}(G)\ :=\ \{\ s \xrightarrow{a} t \mid a \in T\ \wedge\ \exists\ u \in h(a),\ s \underset{G}{\overset{u}{\Longrightarrow}} t\ \}$$

A relation $h \subseteq T \times P$ can be seen as a mapping from T into 2^P associating the image $h(a)$ of any $a \in T$. When $P = S^*$ for some alphabet S, such a mapping is extended by morphism to a *substitution* from T^* into S^* *i.e.* a mapping h from T^* into 2^{S^*} such that $h(\varepsilon) = \{\varepsilon\}$ and $h(uv) = h(u)h(v)$ for every $u, v \in T^*$. The composition of functions is the composition of their relations: $(g \circ h)(a) = h(g(a))$. Taking alphabets O, P, Q, a Q-graph G, and mappings $h \subseteq P \times Q^*$ and $g \subseteq O \times P^*$ such that $\varepsilon \notin g(O)$, we have

$$g^{-1}(h^{-1}(G))\ =\ (g \circ h)^{-1}(G) \tag{1}$$

Another basic property is the commutation between the inverse mapping and the restriction to particular sets. A set C is *stable* in a graph G when any path between vertices in C contains only vertices in C:

$$s_0 \underset{G}{\longrightarrow} s_1 \ldots s_{n-1} \underset{G}{\longrightarrow} s_n\ \wedge\ s_0, s_n \in C\ \Longrightarrow\ s_1, \ldots, s_{n-1} \in C$$

For any stable set C in a P-graph and any mapping h into 2^P, we have

$$h^{-1}(G_{|C})\ =\ (h^{-1}(G))_{|C} \tag{2}$$

Another way to express a restriction of an inverse mapping of a graph is to use a marking of the graph. The *marking* $\#_{C}(G)$ on a vertex set C of a graph G by a symbol $\#$ is the graph:

$$\#_{C}(G)\ :=\ G\ \cup\ \{\ s \xrightarrow{\#} s \mid s \in C\ \}$$

obtained from G by adding $\#$ to any vertex in C. Any restriction of an inverse of a graph is an inverse of a marking of the graph:

$$(h^{-1}(G))_{|C}\ =\ g^{-1}(\#_{C}(G))\quad \textit{where}\ \ g(a)\ =\ \#h(a)\# \tag{3}$$

For ε the neutral element of the free monoid T^* generated by an alphabet T, the *ε-closure* $\overline{G}$ of any $T \cup \{\varepsilon\}$-graph G is obtained by removing the ε-transitions and by adding T-transitions as follows:

$$\overline{G}\ :=\ (Id_{T})^{-1}(G)\ \ =\ \ \{\ s \xrightarrow{a} t \mid a \in T\ \wedge\ s \underset{G}{\overset{a}{\Longrightarrow}} t\ \}$$

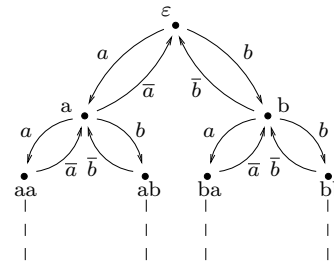
We compare graphs by isomorphism. A *partial isomorphism* from a graph G into a graph H is an injective function such that $s \underset{G}{\overset{a}{\longrightarrow}} t\ \iff\ h(s) \underset{H}{\overset{a}{\longrightarrow}} h(t)$. An *isomorphism* is a partial isomorphism such that $V_G \subseteq Dom(h)$ and $V_H \subseteq Im(h)$. A partial isomorphism on $T \cup \{\varepsilon\}$-graphs considers the ε label as a new letter. To take an ε-transition as an internal (silent) move, we compare $T \cup \{\varepsilon\}$-graphs by partial weak isomorphism. A *partial weak isomorphism* from a graph G into a graph H is an injective function such that $s \underset{G}{\overset{a}{\Longrightarrow}} t\ \iff\ h(s) \underset{H}{\overset{a}{\Longrightarrow}} h(t)$. A *weak isomorphism* is a partial weak isomorphism such that $V_G \subseteq Dom(h)$ and $V_H \subseteq Im(h)$. Note that h is a (resp. partial) weak isomorphism from G into H if and only if H is a (resp. partial) isomorphism from $\overline{G}$ into $\overline{H}$.

3 Classes of Graphs

A general way to define a family REC_F of graphs from a family F of languages is to take the set of graphs obtained from the binary tree (with inverse transitions) by applying an inverse F-mapping followed by a rational restriction [8]. An equivalent way is to take the set of graphs obtained from the binary tree by marking a rational vertex set followed by an inverse F-mapping (Proposition 3.1). We deduce known results on the family REC_{Fin} (Theorem 3.2), on the family REC_{Rat} (Theorem 3.3), on the family $REC_{\overline{L}in}$ where $\overline{L}in$ is a subfamily of linear languages (Theorem 3.5). We deduce also closure properties by inverse mappings.

Let N be an alphabet containing at least two letters. We take a new alphabet $\overline{N}:=\{\ \overline{a}\mid a\in N\ \}$ in bijection with N. We define the following *Dyck graph*:

$\Lambda_N\ :=\ \{\ u\xrightarrow{a}ua\mid u\in N^*\ \wedge\ a\in N\ \}\ \cup\ \{\ ua\xrightarrow{\overline{a}}u\mid u\in N^*\ \wedge\ a\in N\ \}$
where a representation for $N=\{a,b\}$ is the following:



When L is rational, we say that $\#_{_L}(\Lambda_N)$ is a *rational marking* of Λ_N. Let $Dyck_{_N}=[\Lambda_N]$ where $[G]$ is the set of all graphs isomorphic to a graph G, that we extend by union to any class of graphs.
We restrict here a *language family* F to be a subset of $2^{N\cup\overline{N}}$. A family of languages defines a set of mappings: a mapping h is rational (resp linear, ...) if for any letter a, the language $h(a)$ is rational (resp linear, ...). Precisely, a language family F and an alphabet T produce the set F_T of mappings defined for every $a\in T$ by a language $h(a)\in F$. By inverse of a class Φ of $(N\cup\overline{N})$-graphs, we get the following class of T-graphs:

$$F_T^{-1}(\Phi)\ :=\ \{\ h^{-1}(G)\mid G\in\Phi\ \wedge\ h\in F_T\ \}$$

Starting from $Dyck_{_N}$, we have two ways to get classes of graphs. Either we apply inverse F-mappings followed by rational restrictions [8]:

$$F_T^{-1}(Dyck_{_N})_{|}\ :=\ [\{\ h^{-1}(\Lambda_N)_{|\,L}\mid h\in F_T\ \wedge\ L\in Rat(N^*)\ \}]$$

or we apply rational markings followed by inverse F-mappings:

$$F_T^{-1}(\#(Dyck_{_N}))\ :=\ [\{\ h^{-1}(\#_{_L}(\Lambda_N))\mid h\in F_T\ \wedge\ L\in Rat(N^*)\ \}]$$

Henceforth F will be one of the following language families: the family Fin of finite languages; the family Rat of rational languages; the family Lin of linear languages; the family RE of recursively enumerable languages; the subfamily $\overline{L}in$ of linear languages generated by linear grammars such that each right hand side is ε or of the form uBv where B is a non-terminal with $u\in\overline{N}^*$ and $v\in N^*$;

and the rational closure $\overline{L}in(Rat)$ of $\overline{L}in$. For these families, the two previous classes of graphs coincide and we can also restrict N to have only two letters.

Proposition 3.1. *For any distinct letters $a, b \in N$, we have* $F_T^{-1}(Dyck_N)_| = F_T^{-1}(Dyck_{\{a,b\}})_| = F_T^{-1}(\#(Dyck_{\{a,b\}})) = F_T^{-1}(\#(Dyck_N))$ *This class is denoted REC_{F_T} or REC_F when T is understood.*

The class REC_{Fin} is the set of *regular graphs* (see [11], [7]) of bounded degree, and we present again two sets of representatives.

Theorem 3.2. [8] *Given an alphabet N of at least two letters, the following properties are equivalent:*

a) $G \in REC_{Fin_T}$

b) G *is isomorphic to* $(H.N^*)_{|L}$ *for some finite* $H \subseteq N^* \times T \times N^*$ *and* $L \in Rat(N^*)$

c) G *is isomorphic to* $\bigcup_{i=1}^{n}(u_i \xrightarrow{a_i} v_i).W_i$ *for some* $n \geq 0$, $a_1, \ldots, a_n \in T$, $u_1, v_1, \ldots, u_n, v_n \in N^*$, $W_1, \ldots, W_n \in Rat(N^*)$

d) G *is a regular T-graph of bounded degree.*

The traces of the graphs in REC_{Fin} are all the context-free languages.

Recall that a graph $G \subseteq N^* \times T \times N^*$ is *recognizable* if $G = U_1 \xrightarrow{a_1} V_1 \cup \ldots \cup U_n \xrightarrow{a_n} V_n$ for some $n \geq 0, a_1, \ldots, a_n \in T, U_1, V_1, \ldots, U_n, V_n \in Rat(N^*)$. The class REC_{Rat} has been studied in [8] and we present again two sets of representatives.

Theorem 3.3. [8] *Given an alphabet N of at least two letters, the following properties are equivalent:*

a) $G \in REC_{Rat_T}$

b) G *is isomorphic to* $(H.N^*)_{|L}$ *for some recognizable* $H \subseteq N^* \times T \times N^*$ *and* $L \in Rat(N^*)$

c) G *is isomorphic to* $\bigcup_{i=1}^{n}(U_i \xrightarrow{a_i} V_i).W_i$ *for some* $n \geq 0$, $a_1, \ldots, a_n \in T$, $U_1, V_1, W_1 \ldots, U_n, V_n, W_n \in Rat(N^*)$.

The traces of the graphs in REC_{Rat} are all the context-free languages.

Several characterizations of REC_{Fin} inside REC_{Rat} have been given in [2] and [10]. By Proposition 3.1, we get closure properties of these two families.

Proposition 3.4. *We have* $Fin_T^{-1}(REC_{Fin_N}) = REC_{Fin_T}$

and $Rat_T^{-1}(REC_{Fin_N}) = Rat_T^{-1}(REC_{Rat_N}) = REC_{Rat_T}$

It remains to recall the family of rational graphs [14]. We consider a graph as a subset of $N^* \times T \times N^*$ *i.e.* a T-graph with vertices in N^*. We extend the monoid $N^* \times N^*$ to the partial semigroup $N^* \times T \times N^*$ defined by

$$(u, a, v).(x, a, y) = (ux, a, vy)$$

for every $u, v, x, y \in N^*$ and $a \in T$. The extension by union of $.$ to subsets is the usual *synchronization product* for graphs [1]:

$$G.H \;=\; \{\; ux \xrightarrow{a} vy \mid u \xrightarrow[G]{a} v \;\wedge\; x \xrightarrow[H]{a} y \;\} \quad \text{for any} \quad G, H \subseteq N^* \times T \times N^*$$

To this operation is associated the rational family $Rat(N^* \times T \times N^*)$ of graphs: it is the smallest subset of $2^{N^* \times T \times N^*}$ containing the finite graphs and closed by $\cup$, $.$, $+$. A *rational graph* is a graph isomorphic to a graph in $Rat(N^* \times T \times N^*)$; we denote by $RAT_{_T}$ the family of rational T-graphs. The rational graphs are the graphs recognized by the labelled transducers. Precisely, a *T-labelled transducer* is a finite $(N^* \times N^*)$-automaton $A = (G, i, (F_a)_{a \in T})$ with a set F_a of final states for each $a \in T$; such an automaton recognizes the graph:

$$L(A) \;:=\; \{\; u \xrightarrow{a} v \mid \exists\, s \in F_a,\; i \overset{u/v}{\underset{G}{\Longrightarrow}} s \;\}$$

The family $\overline{Lin}$ defines by inverse mappings the class of rational graphs.

Theorem 3.5. **[14]** *We have* $RAT_{_T} \;=\; REC_{\overline{Lin}_T} \;\subset\; REC_{Lin_{_T}}$.
The traces of the graphs in RAT *are context-sensitive languages.*

A conjecture is that the traces of the graphs in RAT are exactly the context-sensitive languages. Note that we can have non recursive traces for graphs in REC_{Lin} . From the closure by composition of rational relations, the rational graphs are closed by inverse finite mappings.

Proposition 3.6. *We have* $Fin_T^{-1}(RAT_{_N}) \;=\; RAT_{_T}$.

We will now use Turing machines to define a general class of graphs whose the traces are the recursively enumerable languages.

4 Graphs of Rewriting Systems and of Turing Machines

We consider the rational restrictions of the ε-closure for the set of transitions of the labelled Turing machines. We show that this family is the same that for the labelled word rewriting systems (Theorem 4.4). We show also that this family is REC_F for any family F of recursively enumerable languages containing the rational closure of the linear languages (Theorem 4.5). Finally, we show that this family is the set of the inverse rational mappings of the rational graphs (Theorem 4.6).

The notion of a word-rewriting system is well-known (see for instance the survey [12] and [4]): it is just a finite set of rules between words. As for the transitions of a pushdown automaton, we allow labelled rules, and to any system, we associate a rational language of admissible words, usually called configurations, which are the words where the rules can be applied. The words are over an alphabet (finite set of symbols) N of *non-terminals*, and the rules are labelled by symbols in an alphabet T of *terminals*, plus the empty word ε.

Definition 4.1. *A finite labelled word rewriting system* (R, C) *is a couple of a finite relation* $R \subseteq N^* \times (T \cup \{\varepsilon\}) \times N^*$ *and a rational language* $C \subseteq N^*$ *of configurations. We write shortly* R *instead of* (R, N^*).

The set of transitions of R is the following $(T \cup \{\varepsilon\})$-graph:

$$T(R) := \{\ xuy \xrightarrow{a} xvy \mid (u, a, v) \in R \ \wedge\ x, y \in N^*\ \}$$

The unlabelled transitions of $T(R)$ form the usual *rewriting* $\underset{R}{\longrightarrow}$ of R:

$$xuy \underset{R}{\longrightarrow} xvy \text{ for some } (u, a, v) \in R \text{ with } x, y \in N^*$$

Its reflexive and transitive closure $\underset{R}{\longrightarrow}^*$ by composition is the *derivation* of R.

To any system (R, C), we associate its *transition graph*:

$$G(R, C) := \overline{T(R)}_{|C} \ =\ \{\ u \xrightarrow{a} v \mid u \underset{T(R)}{\overset{a}{\Longrightarrow}} v \ \wedge\ u, v \in C \ \wedge\ a \in T\ \}$$

which is the restriction to C of the ε-closure of $T(R)$. In particular $G(R) = \overline{T(R)}$.

For instance, the transition relation of a *pushdown automaton* over a set Q of states and over a disjoint set P of stack letters, can be seen as a rewriting system (R, C) over $N = P \cup Q$ where R is in $Q.P \times (T \cup \{\varepsilon\}) \times Q.P^*$, and C is a rational subset of $Q.P^*$. The closure by isomorphism $[G(R, C)]$ of their transition graphs form the family REC_{Rat}.

Another particular rewriting systems are the Turing machines with a read only input tape and a working tape [13], [16]. More exactly and given a set Q of states, a disjoint set T of input tape letters, and a disjoint set $P_\Box = P \cup \{\Box\}$ of working tape letters, a *Turing machine* (M, C) is a finite set M of rules of the form:

$$pA \xrightarrow{a} qB\delta \ \text{ where } \ p, q \in Q,\ a \in T \cup \{\varepsilon\},\ A, B \in P_\Box\ ,\ \delta \in \{+, -\}$$

with a rational set C of configurations $C \in Rat((Q \cup P_\Box)^*)$.

However we are only interested to configurations upv where $p \in Q$ and $u, v \in {P_\Box}^*$ with $u(1), v(|v|) \neq \Box$. Precisely a configuration is of the form $]u]\, p\, [v[$ where for any word $u \in {P_\Box}^*$, $[u[$ (resp. $]u]$) is the greatest prefix (resp. suffix) of u having its last (resp. first) letter distinct of $\Box$ *i.e.* by induction,

$$[u\Box[= [u[\ \wedge\ [u[= u \ \text{ if } u(|u|) \neq \Box \ \text{ and } \ ]\Box u] =]u] \ \wedge\ ]u] = u \ \text{ if } u(1) \neq \Box$$

The set of transitions of M is the following $(T \cup \{\varepsilon\})$-graph:

$$T(M) := \{\ ]u]\, p\, [Av[\xrightarrow{a}]uB]\, q\, [v[\mid pA \underset{M}{\xrightarrow{a}} qB+ \wedge\ u, v \in {P_\Box}^*\ \}$$

$$\cup\ \{\ ]uC]\, p\, [Av[\xrightarrow{a}]u]\, q\, [CBv[\mid pA \underset{M}{\xrightarrow{a}} qB- \wedge\ C \in P_\Box \ \wedge\ u, v \in {P_\Box}^*\ \}$$

Hence the *transition graph* of any Turing machine (M, C) is the T-graph:

$$G(M, C) \ :=\ \overline{T(M)}_{|C}$$

The transition graph of any Turing machine is the transition graph of a *stable* rewriting system (R, C) meaning that C is stable in $T(R)$:

$$s \underset{R}{\longrightarrow}^* r \underset{R}{\longrightarrow}^* t \ \wedge\ s, t \in C \ \implies\ r \in C$$

Lemma 4.2. *We can transform any Turing machine* M *into a stable rewriting system* (R, C) *such that* $T(R)_{|C}$ *is isomorphic to* $T(M)$.

Proof. We take a new symbol \$ and the following rational language:

$$C \ =\ \{\ \$upv\$ \mid p \in Q \ \wedge\ u, v \in {P_\Box}^* \ \wedge\ u(1), v(|v|) \neq \Box\ \}$$

We transform any rule $pA \xrightarrow{a} qB+$ of M into the following rules:

$$\left|\begin{array}{ll} CpA \xrightarrow{a} CBq & \text{if } CB \neq \$\Box \\ Cp\$ \xrightarrow{a} CBq\$ & \text{if } A = \Box \ \wedge \ CB \neq \$\Box \\ \$pA \xrightarrow{a} \$q & \text{if } B = \Box \\ \$p\$ \xrightarrow{a} \$q\$ & \text{if } A = B = \Box \end{array}\right.$$

We transform any rule $pA \xrightarrow{a} qB-$ of M into the following rules:

$$\left|\begin{array}{ll} CpAD \xrightarrow{a} qCBD & \text{if } C \neq \$ \ \wedge \ BD \neq \Box\$ \\ CpA\$ \xrightarrow{a} qC\$ & \text{if } B = \Box \ \wedge \ C \neq \Box, \$ \\ \Box pA\$ \xrightarrow{a} q\$ & \text{if } B = \Box \\ \\ Cp\$ \xrightarrow{a} qCB\$ & \text{if } A = \Box \neq B \ \wedge \ C \neq \$ \\ Cp\$ \xrightarrow{a} qC\$ & \text{if } A = B = \Box \ \wedge \ C \neq \Box, \$ \\ \Box p\$ \xrightarrow{a} q\$ & \text{if } A = B = \Box \\ \\ \$pAD \xrightarrow{a} \$q\Box BD & \text{if } BD \neq \Box\$ \\ \$pA\$ \xrightarrow{a} \$q\$ & \text{if } B = \Box \\ \$p\$ \xrightarrow{a} \$q\Box B\$ & \text{if } A = \Box \neq B \\ \$p\$ \xrightarrow{a} \$q\$ & \text{if } A = B = \Box \end{array}\right.$$

In this way, we obtain a system R such that C is closed by $\underset{R}{\longrightarrow}$ and

$$U \xrightarrow[T(M)]{a} V \iff \$U\$ \xrightarrow[T(R)]{a} \$V\$ \ \wedge \ \$U\$, \$V\$ \in C$$

Thus (R, C) is stable and $\$T(M)\$ = T(R)_{|C}$.
□

Conversely and up to the ε-transitions, any rewriting system can be simulated by a Turing machine.

Lemma 4.3. *We can transform any rewriting system R into a Turing machine (M, C) such that $\overline{T(M)}_{|C}$ is isomorphic to $\overline{T(R)}$.*

Proof. We denote by m_1 (resp. m_2) the maximum length of the left (resp. right) hand sides of the rules of R *i.e.*

$$m_1 = max\{\ |U| \mid \exists\ a, V,\ (U, a, V) \in R\ \}$$
$$\text{and } m_2 = max\{\ |V| \mid \exists\ U, a,\ (U, a, V) \in R\ \}$$

We take two new symbols $\bullet$ and \$, and we define the following state set Q of the Turing machine to be constructed:

$$Q = \{\bullet\} \ \cup \ N^{\leq m_1} \times (N^{\leq m_2} \cup \{\$\}) \times (T \cup \{\varepsilon\})$$

We take the following set M' of Turing rules:

$$M' \left|\begin{array}{ll} \bullet A \xrightarrow{\varepsilon} \bullet A+ & \text{for } A \in N_\Box \\ \bullet A \xrightarrow{\varepsilon} \bullet A- & \text{for } A \in N_\Box \\ \bullet A \xrightarrow{\varepsilon} (U, V, a)A+ & \text{for } A \in N_\Box \text{ and } (U, a, V) \in R \\ \\ (AU, BV, a)A \xrightarrow{\varepsilon} (U, V, a)B+ & \\ \\ (\varepsilon, BV, a)A \xrightarrow{\varepsilon} (\varepsilon, VA, a)B+ & \text{for } A \in N \\ (\varepsilon, BV, a)\Box \xrightarrow{\varepsilon} (\varepsilon, V, a)B+ & \\ \\ (AU, \varepsilon, a)A \xrightarrow{\varepsilon} (UB, \varepsilon, a)B+ & \text{for } B \in N \\ (AU, \varepsilon, a)A \xrightarrow{\varepsilon} (U, \$, a)\Box+ & \\ (AU, \$, a)A \xrightarrow{\varepsilon} (U, \$, a)\Box+ & \\ (\varepsilon, \$, a)\Box \xrightarrow{\varepsilon} (\varepsilon, \varepsilon, a)\Box+ & \end{array}\right.$$

In this way, we obtain a Turing machine M' such that for every $a \in T \cup \{\varepsilon\}$ and for every $U, V \in N^*$,

$$U \xrightarrow[T(R)]{a} V \iff {}^{\bullet}U \overset{\varepsilon}{\underset{T(M')}{\Longrightarrow}} X(\varepsilon,\varepsilon,a)Y \ \wedge \ [XY[= V$$

We complete M' to M by adding the following rules:

$$\Big| \ (\varepsilon,\varepsilon,a)A \xrightarrow{a} {}^{\bullet}A+ \text{ for } A \in N_{\square}$$

The relation $h = \{ (U, {}^{\bullet}U) \mid U \in N^* \}$ is a partial weak isomorphism from $T(R)$ into $T(M)$. More precisely and for every $a \in T \cup \{\varepsilon\}$, we have

$$U \xrightarrow[T(R)]{a} V \implies {}^{\bullet}U \overset{a}{\underset{T(M)}{\Longrightarrow}} {}^{\bullet}V \text{ and } {}^{\bullet}U \overset{a}{\underset{T(M)}{\Longrightarrow}} {}^{\bullet}V \implies U \xrightarrow[T(R)]{a} V$$

So h is a partial isomorphism from $\overline{T(R)}$ into $\overline{T(M)}$. Thus h is a partial isomorphism from $\overline{T(R)}$ into $\overline{T(M)}_{|C}$ where $C = Im(h) = {}^{\bullet}N^*$.

As $V_{\overline{T(R)}} \subseteq N^* = Dom(h)$, the graphs $\overline{T(R)}$ and $\overline{T(M)}_{|C}$ are isomorphic.
□

The rewriting systems and the Turing machines have the same transition graphs.

Theorem 4.4. *The Turing machines and the rewriting systems define up to isomorphism, the same family of transition graphs, and the traces are the recursively enumerable languages.*

Proof. **i)** Let (M, D) be a Turing machine.
By Lemma 4.2, we can construct a stable rewriting system (R, C) and an isomorphism h from $T(M)$ to $T(R)_{|C}$.
By restriction, h defines an isomorphism from $\overline{T(M)}$ to $\overline{T(R)_{|C}}$.
By Equation (2), $\overline{T(R)_{|C}} = \overline{T(R)}_{|C} = G(R, C)$. Thus $G(M, D) = \overline{T(M)}_{|D}$ is isomorphic (by a restriction of h) to $\overline{T(R)}_{|C \cap h(D)} = G(R, C \cap h(D))$.

ii) Let (R, C) be a rewriting system.
By Lemma 4.3, we can construct a Turing machine (M, D) and an isomorphism h from $\overline{T(R)}$ to $\overline{T(M)}_{|D}$. Thus $G(R, C) = \overline{T(R)}_{|C}$ is isomorphic (by a restriction of h) to $\overline{T(M)}_{|h(C)\cap D} = G(M, h(C) \cap D)$.
□

We denote by $TURING_T$ the family of T-graphs isomorphic to the transition graphs of Turing machines (or of rewriting systems). As for the previous graph families (investigated in the previous section), we characterize the family $TURING_T$ by inverse mappings of the binary tree. The images of these mappings can be the class of recursively enumerable languages, or can be only the class of the rational closure $\overline{L}in(Rat)$ of $\overline{L}in$.

Theorem 4.5. *We have* $TURING_T = REC_{\overline{L}in(Rat)_T} = REC_{RE_T}$

Proof. **i)** Let us show that $TURING_T \subseteq REC_{\overline{L}in(Rat)_T}$.
Let (R, C) be a rewriting system: R is a finite subset of $N^* \times (T \cup \{\varepsilon\}) \times N^*$ and C is a rational subset of N^*.

We replace in R the label ε by a new letter \$:

$$S \;:=\; \{\; (u,a,v) \in R \mid a \in T \;\}\; \cup\; \{\; (u,\$,v) \mid (u,\varepsilon,v) \in R \;\}$$

So $T(S) = h^{-1}(\Lambda_N)$ where h is the following linear mapping:

$$h(a) \;=\; \{\; \widetilde{\overline{x}}\,\widetilde{\overline{u}}\,v\,x \mid (u,a,v) \in S \;\wedge\; x \in N^* \;\} \quad \text{for every} \quad a \in T \cup \{\$\}$$

Furthermore $\overline{T(R)} \;=\; g^{-1}(T(S))$ where g is the following rational mapping:

$$g(a) \;=\; \$^* a \$^* \quad \text{for every} \quad a \in T$$

By (1), we have $\overline{T(R)} = (g \circ h)^{-1}(\Lambda_N)$ where $g \circ h$ is the following mapping:

$$(g \circ h)(a) \;=\; h(g(a)) \;=\; h(\$^* a \$^*) \;=\; h(\$)^* h(a) h(\$)^* \;\in\; \overline{Lin}(Rat)$$

Finally and by Proposition 3.1, $G(R,C) \;=\; \overline{T(R)}_{|C} \;\in\; REC_{\overline{Lin(Rat)}_T}$.

ii) $REC_{\overline{Lin(Rat)}} \;\subseteq\; REC_{RE}$ because $\overline{Lin}(Rat) \;\subseteq\; RE$.

iii) Let us show that $REC_{RE_T} \;\subseteq\; TURING_T$.

Let a mapping $h : \; T \longrightarrow RE(\{a,b,\overline{a},\overline{b}\}^*)$. By Proposition 3.1 , it is sufficient to construct a rewriting system (R,C) such that $h^{-1}(\Lambda_{\{a,\,b\}})$ is isomorphic to $\overline{T(R)}_{|C}$.

For every $c \in T$, there is a Turing machine M_c : a finite set of rules of the form:

$pA \xrightarrow{x} qB\delta$ where $p,q \in Q_c$, $x \in \{\varepsilon,a,b,\overline{a},\overline{b}\}$, $A,B \in P_c \cup \{\Box\}$, $\delta \in \{+,-\}$

plus an initial configuration i_c and a set $F_c \subseteq Q_c$ of final states recognizing:

$$h(c) \;=\; L(T(M_c), i_c, \{]u][q[v[\mid q \in F_c \;\wedge\; u,v \in (P_c \cup \Box)^* \;\})$$

Up to renaming, we may assume that the sets $(P_c)_{c\in T}, (Q_c)_{c\in T}$ are pairwise disjoints, and we define the following Turing machine:

$$M = \{\; pA \xrightarrow{\varepsilon} qB\delta \;\in M_c \mid c \in T \;\}$$
$$\cup\; \{\; pA \xrightarrow{\varepsilon} q_x B\delta \mid \exists\, c \in T,\; pA \xrightarrow{x} qB\delta \;\in M_c \;\wedge\; x \neq \varepsilon \;\}$$

We take three new symbols $\$, \&, \bullet$ and we construct a rewriting system R. First, we take the following rules:

$$\$\$ \xrightarrow{\varepsilon} \$ i_c \$ \quad \text{for every} \quad c \in T$$

to describe the moves between two \$ of the Turing machines defining h. We transform (as in Lemma 4.2) any rule $pA \xrightarrow{\varepsilon} qB+$ of M into the following rules:

$$\begin{array}{rll} CpA \xrightarrow{\varepsilon} CBq & \text{if} & CB \neq \$\Box \\ Cp\$ \xrightarrow{\varepsilon} CBq\$ & \text{if} & A = \Box \;\wedge\; CB \neq \$\Box \\ \$pA \xrightarrow{\varepsilon} \$q & \text{if} & B = \Box \\ \$p\$ \xrightarrow{\varepsilon} \$q\$ & \text{if} & A = B = \Box \end{array}$$

In a same way (and as in the proof of Lemma 4.2), we transform any rule $pA \xrightarrow{\varepsilon} qB-$ of M into new rules of R.

For every $c \in T$, $q \in Q_c$, $A \in P_c \cup \{\Box\}$, $y \in \{a,b,\overline{a},\overline{b}\}$, $x \in \{a,b\}$, we take the rules:

$$\begin{array}{r} q_y \xrightarrow{\varepsilon} q'_y \& \\ A q'_y \xrightarrow{\varepsilon} q'_y A \\ x\$q'_{\overline{x}} \xrightarrow{\varepsilon} \$\overline{q} \\ \$q'_x \xrightarrow{\varepsilon} x\$\overline{q} \\ \overline{q}A \xrightarrow{\varepsilon} A\overline{q} \\ \overline{q}\& \xrightarrow{\varepsilon} q \end{array}$$

For the acceptance and for every $c \in T$ and $A \in (\bigcup_c P_c) \cup \{\Box\}$, we define

$$\left|\begin{array}{ll} q \xrightarrow{c} \bullet & \text{if } q \in F_c \\ \bullet A \xrightarrow{\varepsilon} \bullet & \\ A\bullet \xrightarrow{\varepsilon} \bullet & \\ \$\bullet\$ \xrightarrow{\varepsilon} \$\$ & \end{array}\right.$$

For every $c \in T$ and $u, v \in \{a,b\}^*$, we have

$$u \xrightarrow[h^{-1}(\Lambda_{\{a,b\}})]{c} v \iff u\$\$ \overset{c}{\underset{T(R)}{\Longrightarrow}} v\$\$$$

So $h = \{ (u, u\$\$) \mid u \in \{a,b\}^* \}$ is a partial weak isomorphism from $h^{-1}(\Lambda_{\{a,b\}})$ into $T(R)$. Thus h is a partial isomorphism from $\overline{h^{-1}(\Lambda_{\{a,b\}})} = h^{-1}(\Lambda_{\{a,b\}})$ into $\overline{T(R)}$. Hence h is an isomorphism from $h^{-1}(\Lambda_{\{a,b\}})$ into $\overline{T(R)}_{|C}$ where $C = Im(h) = \{a,b\}^*\$\$$.
□

The class $TURING$ is the closure of RAT by inverse rational mapping.

Theorem 4.6. *We have* $Rat_T^{-1}(RAT_N) = TURING_T$.

Proof. **i)** $TURING_T \subseteq Rat_T^{-1}(RAT_N)$.

Let $G \in TURING_T$: G is isomorphic to $\overline{T(R)}_{|C}$ for some rewriting system (R, C). Let $\#, \$$ be two new symbols. We have

$$\overline{T(R)} = h^{-1}(T(S))$$

where h is the rational mapping defined by $h(a) = \$^* a \$^*$ for every $a \in T$, and S is the system obtained from R by replacing the label ε by $\$$:

$$S := \{ (u, a, v) \in R \mid a \in T \} \cup \{ (u, \$, v) \mid (u, \varepsilon, v) \in R \}$$

By Equations (3) and (1), we have

$$\overline{T(R)}_{|C} = h_{\#}^{-1}\Big(T(S) \cup \{u \xrightarrow{\#} u \mid u \in C\}\Big)$$

where $h_{\#}(a) = \#\$^* a \$^*\#$ for every $a \in T$.

Obviously $T(S)$ is a rational graph and $\{u \xrightarrow{\#} u \mid u \in C\}$ is also a rational graph because C is a rational language. Hence $G \in Rat_T^{-1}(RAT_N)$.

ii) $Rat_T^{-1}(RAT_N) \subseteq TURING_T$.

Let $G \in Rat_T^{-1}(RAT_N)$: there is a rational N-graph H and a mapping h from T into $Rat(N^*)$ such that $G = h^{-1}(H)$.

By definition of a rational graph, there is an alphabet X and a *N-labelled transducer* $A = (K, i, (E_x)_{x \in N})$ where K is a finite $(X^* \times X^*)$-automaton, i is the initial state, and for each $x \in N$, E_x is a set of final states, and such that the automaton A recognizes the graph $L(A) = \{ u \xrightarrow{x} v \mid \exists\, s \in E_x,\ i \overset{u/v}{\underset{K}{\Longrightarrow}} s \}$ isomorphic to H.

Furthermore and for each $a \in T$, there is a finite N-automaton (K_a, i_a, F_a) recognizing the rational language $h(a)$.

We may assume that the automata $(K_a)_{a \in T}$ have pairwise disjoint state sets: $V_{K_a} \cap V_{K_b} = \emptyset$ for $a \neq b$. We denote by $\overline{K} = \bigcup_{a \in T} K_a$ and we take a new state $\overline{\imath} \notin V_{\overline{K}}$.

We take a new symbol \$ and we denote by $C = \$\bar{\imath}X^*\$$ the (rational) configuration set of the following rewriting system R:

$$R \left| \begin{array}{rll} \bar{\imath} \xrightarrow{\varepsilon} i_a & \text{for each} & a \in T \\ \$s \xrightarrow{\varepsilon} \$(i,s) & \text{for each} & s \in V_{\overline{K}} \\ \$s \xrightarrow{a} \$\bar{\imath} & \text{if} & s \in F_a \\ (p,s)u \xrightarrow{\varepsilon} v(q,s) & \text{if} & p \xrightarrow[K]{u/v} q \ \text{ and } \ s \in V_{\overline{K}} \\ (p,s)\$ \xrightarrow{\varepsilon} t\$ & \text{if} & p \in E_x \ \text{ and } \ s \xrightarrow[\overline{K}]{x} t \ \text{ for some } \ x \in N \\ As \xrightarrow{\varepsilon} sA & \text{for each} & A \in X \ \text{ and } \ s \in V_{\overline{K}} \end{array} \right.$$

Thus
$$\begin{aligned} \overline{T(R)}_{|C} &= \{\ \$\bar{\imath}u\$ \xrightarrow{a} \$\bar{\imath}v\$ \mid \$\bar{\imath}u\$ \underset{T(R)}{\overset{a}{\Longrightarrow}} \$\bar{\imath}v\$\ \} \\ &= \{\ \$\bar{\imath}u\$ \xrightarrow{a} \$\bar{\imath}v\$ \mid \exists\ s \in F_a \ \ \$i_a u\$ \underset{T(R)}{\overset{\varepsilon}{\Longrightarrow}} \$sv\$\ \} \\ &= \{\ \$\bar{\imath}u\$ \xrightarrow{a} \$\bar{\imath}v\$ \mid \exists\ w \in h(a) \ \ u \underset{L(A)}{\overset{w(1)}{\longrightarrow}} \cdots \underset{L(A)}{\overset{w(|w|)}{\longrightarrow}} v\ \} \\ &= \{\ \$\bar{\imath}u\$ \xrightarrow{a} \$\bar{\imath}v\$ \mid u \underset{L(A)}{\overset{h(a)}{\Longrightarrow}} v\ \} \\ &= \$\bar{\imath}h^{-1}(L(A))\$ \quad \text{isomorphic to} \quad h^{-1}(H) = G\,. \end{aligned}$$

□

In particular RAT is not closed by inverse rational mapping. We also deduce that the transition graphs of Turing machines are the rational restrictions of the ε-closure of rational graphs (with ε-arcs).

5 Conclusion

We have presented a hierarchy of graph families and essentially the family $TURING$ of transition graphs of labelled Turing machines with ε-rules. In particular REC_{Rat} is the family of transition graphs of pushdown automata with ε-rules. Between the lowest family FIN of finite graphs and the greatest family $TURING$, we can show that the two families REC_{Rat} and RAT are natural, by considering the Cayley graphs of the word rewriting systems [6], [9] (and a work in preparation with Knapik). Finally and using traces, the hierarchy FIN , REC_{Rat} , RAT , $TURING$ yields a Chomsky hierarchy, except that it remains to see whether any context-sensitive language is the trace of a rational graph. Another point is to find a subclass of word rewriting systems such that their transition graphs are the graphs of REC_{Lin} .

References

1. A. Arnold and M. Nivat *Comportements de processus*, Colloque AFCET "Les mathématiques de l'informatique", 35–68 (1982).
2. K. Barthelmann *When can an equational simple graph be generated by hyperedge replacement*, MFCS 98, LNCS 1450, L. Brim, J. Gruska, J. Zlatuska (Eds.), 543–552 (1998).

3. J. Berstel *Transductions and context-free languages*, Teubner (Ed.), Stuttgart (1979).
4. R. Book and F. Otto *String-rewriting systems*, Texts and monographs in computer science, P. Gries (Ed.), Springer-Verlag (1993).
5. R. Büchi *Regular canonical systems*, Archiv für Mathematische Logik und Grundlagenforschung 6, 91–111 (1964) [reprinted in *The collected works of J. Richard Büchi*, S. Mac Lane, D. Siefkes (Eds.), Springer-Verlag, New York, 317–337 (1990)].
6. H. Calbrix and T. Knapik *A string-rewriting characterization of Muller and Schupp's context-free graphs*, FSTTCS 98, LNCS 1530, 331–342 (1998).
7. D. Caucal *On the regular structure of prefix rewriting*, CAAP 90, LNCS 431, A. Arnold (Ed.), 87–102 (1990) [a full version is in Theoretical Computer Science 106, 61–86 (1992)].
8. D. Caucal *On infinite transition graphs having a decidable monadic theory*, ICALP 96, LNCS 1099, F. Meyer auf der Heide, B. Monien (Eds.), 194–205 (1996) [a full version will appear in Theoretical Computer Science].
9. D. Caucal *On word rewriting systems having a rational derivation*, FOSSACS 00, LNCS 1784, J. Tiuryn (Ed.), 48–62 (2000).
10. D. Caucal and T. Knapik *An internal presentation of regular graphs by prefix-recognizable ones*, accepted for publication in Theory of Computing Systems.
11. B. Courcelle *Graph rewriting: an algebraic and logic approach*, Handbook of Theoretical Computer Science Vol. B, J. Leeuwen (Ed.), Elsevier, 193–242 (1990).
12. N. Dershowitz and J.-P. Jouannaud *Rewrite systems*, Handbook of Theoretical Computer Science Vol. B, J. Leeuwen (Ed.), Elsevier, 243–320 (1990).
13. A. Mateescu and A. Salomaa *Aspects of classical language theory*, Handbook of Formal Languages Vol. 1, G. Rozenberg, A. Salomaa (Eds.), Springer-Verlag, 175–251 (1997).
14. C. Morvan *On rational graphs*, FOSSACS 00, LNCS 1784, J. Tiuryn (Ed.), 252–266 (2000).
15. D. Muller and P. Schupp *The theory of ends, pushdown automata, and second-order logic*, Theoretical Computer Science 37, 51–75 (1985).
16. E. Payet *Produit synchronisé pour quelques classes de graphes infinis*, PhD Thesis, University of La Réunion (2000).
17. T. Urvoy *Regularity of congruential graphs*, MFCS 00, LNCS 1893, M. Nielsen and B. Rovan (Eds.), 680–689 (2000).

JC-Nets

Gabriel Ciobanu and Mihai Rotaru

Institute of Theoretical Computer Science
Romanian Academy, Iaşi, Romania
gabriel@info.uaic.ro
mrotaru@iit.tuiasi.ro

Abstract. In this paper we introduce and study a graphical representation of the join calculus given by the so called jc-nets. jc-nets provide a fully abstract semantics for a Turing complete fragment of the join calculus. Thus a new formal model for concurrency and distributed systems is introduced. This new formal model simplifies Milner's π-nets, while preserving their expressive power.

1 Introduction

New communication technologies are changing the landscape of computing. The Internet has become a platform for large scale distributed programming. Turing's model of computation presents programs as computing functions or relations from inputs to outputs. Nowadays we work with new concepts like multitasking, distributed systems, mobile agents ... that have appeared since we deal with global computation instead of isolated systems with few interactions with their environment. The key feature of this new computation paradigm is that information is flowing around and programs interact each other. Complex behaviours arise as the global effect of a system of interacting processes. We need new conceptual tools, new models and theories to describe and understand, to design and build up these new systems.

Currently there are several languages for programming distributed systems. Many of these languages lack a complete formal definition and, in consequence, they lack rigorous techniques to reason about program equivalence and program properties. We refer in this paper to the Join Calculus Language together with its core part called the join calculus. The join calculus could be considered as a version of the π-calculus and it preserves its expressive power. The π-calculus [Mil93,Mil99] is a term algebra able to express mobility. Interaction in the π-calculus is given by shared names used as I/O channels. The π-calculus works with channels, guards and messages. In the graphical presentation of the π-terms given by the π-nets [Mil94], channels are represented as complex nodes called torpedos, guards are represented as boxes and messages are represented as directed arcs. In this graphical representation, the boxes hide the internal nodes which represent channels. To make available the hidden channels, Milner uses a rather complex auxiliary mechanism of links. In order to avoid this auxiliary mechanism (that has no corresponding elements in the π-calculus), we

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 190–201, 2001.

designed a new mapping such that channels are represented by simple nodes, messages are represented by boxes of arcs and guards are represented by arcs between boxes. This new mapping simplifies the graphical representation of the π-calculus. Unfortunately it leads to identical representations for processes with different behaviours. Interestingly enough, this mapping works well when we deal with the join calculus. Using this mapping we introduce and study a graphical formalism called `jc`-nets. We use these `jc`-nets to interpret a subset of the join calculus. Therefore we present a new formal model for concurrent and distributed systems that comes with an important structural reduction of the π-nets while preserving their expressive power. The definition of the `jc`-nets is given by the same algebraic framework used for π-nets.

2 The Join Calculus

In this section we recall the definition of the join calculus. The join calculus was introduced by Fournet and Gonthier in [FG96] as a core calculus to underlie a concurrent programming language. A detailed account may be found in [Fou99]. Our presentation is based on [Lev98]. First we consider a restriction of the join calculus. Let $\mathcal{X}$ be an infinite countable set of *names*. The symbols $u, v, x, \ldots$ range over names, and $\tilde{u}, \tilde{v}, \tilde{x}, \ldots$ range over finite sequences of names. We denote by $\mathcal{P}$ the set of the join calculus terms which are called *processes*. The symbols $P, Q, R, \ldots$ range over processes.

Definition 1. Processes *are given by the following grammar:*

$$
\begin{array}{rcll}
P & ::= & 0 & \text{empty process} \\
 & | & u\langle v\rangle & \text{output message} \\
 & | & P \mid Q & \text{parallel composition} \\
 & | & \texttt{def}\ u\langle v\rangle \triangleright P\ \texttt{in}\ Q & \text{definition}
\end{array}
$$

0 is the empty process. An output message $u\langle v\rangle$ denotes the emission of a name v along a channel u. $P \mid Q$ represents the parallel composition of P and Q. Interprocess communication is achieved via definitions. A definition $\texttt{def}\ u\langle v\rangle \triangleright P\ \texttt{in}\ Q$ results in a receptor channel u being created for the interaction between processes P and Q. Process Q could eventually send a channel name y along channel u. At the other end of channel u, process P is waiting for the reception of channel names. When the name y, sent by the process Q, is received from channel u, process Q will continue its execution in parallel with process $\{y/v\}P$ and the receptor channel u is open to receive new names.

In the above syntax, only the definition binds names. Thus u and v are bound, and the scope of v is P, whereas the scope of u extends to the whole definition. α-conversion is defined in the standard way. We write $\{y/x\}P$ to denote the usual capture-avoiding substitution of the free occurrences of name x by name y in the term P. Formally, the *free names* of the processes are defined inductively by:

$$
\texttt{fn}(0) = \emptyset \qquad \texttt{fn}(u\langle v\rangle) = \{u, v\} \qquad \texttt{fn}(P \mid Q) = \texttt{fn}(P) \cup \texttt{fn}(Q)
$$

$$
\texttt{fn}(\texttt{def}\ u\langle v\rangle \triangleright P\ \texttt{in}\ Q) = (\texttt{fn}(Q) \cup (\texttt{fn}(P) - \{v\})) - \{u\}
$$

Definition 2. Structural congruence $\equiv \subseteq \mathcal{P} \times \mathcal{P}$ *is the smallest congruence relation which satisfies the following axioms:*

A1: $\texttt{def}\ u\langle v\rangle \triangleright P\ \texttt{in}\ Q \equiv \texttt{def}\ u\langle t\rangle \triangleright \{t/v\}P\ \texttt{in}\ Q$, *if* $t \notin \texttt{fn}(P)$
A2: $\texttt{def}\ u\langle v\rangle \triangleright P\ \texttt{in}\ Q \equiv \texttt{def}\ w\langle v\rangle \triangleright \{w/u\}P\ \texttt{in}\ \{w/u\}Q$
if $v \notin \{u, w\}$ *and* $w \notin \texttt{fn}(P) \cup \texttt{fn}(Q)$
P1: $P \mid 0 \equiv P$ P2: $P \mid Q \equiv Q \mid P$ P3: $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
D1: $Q_1 \mid \texttt{def}\ u\langle v\rangle \triangleright P\ \texttt{in}\ Q_2 \equiv \texttt{def}\ u\langle v\rangle \triangleright P\ \texttt{in}\ (Q_1 \mid Q_2)$ *if* $u \notin \texttt{fn}(Q_1)$
D2: $\texttt{def}\ u\langle v\rangle \triangleright P_1\ \texttt{in def}\ w\langle t\rangle \triangleright P_2\ \texttt{in}\ Q \equiv \texttt{def}\ w\langle t\rangle \triangleright P_2\ \texttt{in def}\ u\langle v\rangle \triangleright P_1\ \texttt{in}\ Q$
if $u \neq w$, $u \notin \texttt{fn}(P_2)$, *and* $w \notin \texttt{fn}(P_1)$

Definition 3. Reduction $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ *is the smallest relation which satisfies:*

R1 $\texttt{def}\ u_1\langle v_1\rangle \triangleright Q_1\ \texttt{in def}\ u_2\langle v_2\rangle \triangleright Q_2\ \texttt{in}\ \ldots\ \texttt{def}\ u_n\langle v_n\rangle \triangleright Q_n\ \texttt{in}\ (P \mid u_i\langle v\rangle) \rightarrow$
$\texttt{def}\ u_1\langle v_1\rangle \triangleright Q_1\ \texttt{in def}\ u_2\langle v_2\rangle \triangleright Q_2\ \texttt{in}\ \ldots\ \texttt{def}\ u_n\langle v_n\rangle \triangleright Q_n\ \texttt{in}\ (P \mid \{v/v_i\}Q_i)$
if $\{u_{i+1}, \ldots, u_n\} \cap (\texttt{fn}(Q_i) \cup \{u_i\}) = \emptyset$ *where* $i \in [n]$ *and* $n \geq 1$.
R2 $P_1 \rightarrow P_2$ *implies* $\texttt{def}\ u\langle v\rangle \triangleright Q\ \texttt{in}\ P_1 \rightarrow \texttt{def}\ u\langle v\rangle \triangleright Q\ \texttt{in}\ P_2$
R3 $P_1 \equiv Q_1$, $Q_1 \rightarrow Q_2$, *and* $Q_2 \equiv P_2$ *implies* $P_1 \rightarrow P_2$.

We should remark that we have no reduction rule for parallel composition. Such a rule is just a consequence. It is proved that if $P_1 \rightarrow P_2$, then $Q \mid P_1 \rightarrow Q \mid P_2$. Moreover, for any substitution $\sigma = \{y/x\}$, $P_1 \rightarrow P_2$ implies $\sigma P_1 \rightarrow \sigma P_2$.

If in the definition of processes we consider messages of the form $u\langle\tilde{v}\rangle$ and definitions of the form $\texttt{def}\ u\langle\tilde{v}\rangle \triangleright P\ \texttt{in}\ Q$, we get a polyadic version of the join calculus. Fournet shows in [Fou99] that this polyadic version of the join calculus can encode the call-by-name λ-calculus. Consequently this polyadic version is Turing complete. The definitions and results presented in this paper can be extended to the polyadic version of the join calculus in the same way the monadic π-nets are extended to the polyadic π-nets [Mil94]. In the full version of the join calculus the processes are defined by using two new syntactic categories, namely *definitions* and *join-patterns*.

$$\begin{array}{lll} P ::= 0 & \mathrm{D} ::= J \triangleright P & J ::= u\langle\tilde{v}\rangle \\ \quad \mid\ u\langle\tilde{v}\rangle & \quad \mid\ D \wedge D & \quad \mid\ J \mid J \\ \quad \mid\ P \mid P & & \\ \quad \mid\ \texttt{def}\ D\ \texttt{in}\ P & & \end{array}$$

As above, the scopes are determined by terms of the form $\texttt{def}\ D\ \texttt{in}\ P$ that define new channels for the body P where they are used. The set of defined channels in D is denoted by $\texttt{dn}(D)$. A join-pattern J binds a set $\texttt{rn}(J)$ of received values. We also extend the set $\texttt{fn}(P)$ of the free names.

$$\begin{array}{ll} \texttt{rn}(x\langle\tilde{v}\rangle) = \{\tilde{v}\} & \texttt{dn}(x\langle\tilde{v}\rangle) = \{x\} \\ \texttt{rn}(J \mid J') = \texttt{rn}(J) \cup \texttt{rn}(J') & \texttt{dn}(J \mid J') = \texttt{dn}(J) \cup \texttt{dn}(J') \\ \texttt{fn}(J \triangleright P) = \texttt{dn}(J) \cup (\texttt{fn}(P) - \texttt{rn}(J)) & \texttt{dn}(J \triangleright P) = \texttt{dn}(J) \\ \texttt{fn}(D \wedge D') = \texttt{fn}(D) \cup \texttt{fn}(D') & \texttt{dn}(D \wedge D') = \texttt{dn}(D) \cup \texttt{dn}(D') \end{array}$$

$$\texttt{fn}(0) = \emptyset \qquad \texttt{fn}(x\langle\tilde{v}\rangle) = \{x\} \cup \{\tilde{v}\} \qquad \texttt{fn}(P \mid P') = \texttt{fn}(P) \cup \texttt{fn}(P')$$
$$\texttt{fn}(\texttt{def}\ D\ \texttt{in}\ P) = (\texttt{fn}(P) \cup \texttt{fn}(D)) - \texttt{dn}(D)$$

When the sequences of names $\tilde{v}$ have the length 1, then we get the *monadic join calculus*. We don't present here the extension of structural congruence and reduction to full join calculus – see [Lev98] for this extension.

3 The Control Structure of the jc-Nets

3.1 Control Structures

We present the jc-nets as a control structure. First we review the control structures. Most of the ideas on the control structures we present in this subsection can be found in the paper [MMP95]. A detailed account of the control structures may be found in [Mif96].

From an algebraic point of view, a control structure consists in a set of terms together with an equational theory and a reduction relation upon terms called *reaction*. A control structure can be seen as a symmetric strict monoidal category (ssmc for short) with additional structure. The ssmc morphisms denoted by $a, b, c, \ldots$ correspond to the terms of the control structure (in the algebraic setting). They are called *actions*. The ssmc objects denoted by $m, n, k, \ldots$ are called *arities*. The monoid of the arities $(M, \otimes, \epsilon)$ is assumed to be freely generated by a set P. The elements of P, denoted by $p, q, \ldots$ are called *prime* arities. If the arities m and n are the domain and respectively the codomain of a, we write $a : m \to n$; with some abuse of terminology, we say that a has the arity $m \to n$. We write $a \cdot b : m \to k$ for the ssmc composition of $a : m \to n$ and $b : n \to k$, and $a \otimes b : m \otimes k \to n \otimes l$ for the ssmc tensorial product of $a : m \to n$ and $b : k \to l$. By $\mathtt{id}_m : m \to m$ we denote the ssmc identities, and by $\mathtt{p}_{m,n} : m \otimes n \to n \otimes m$ the ssmc symmetries.

A control structure uses a denumerable set $\mathcal{X}$ of *names* denoted by $x, y, z, \ldots$. To define the terms, we need a signature $(P, \mathcal{K})$, where P is the set of prime arities, and $\mathcal{K}$ is a set of *control* operators. Each name $x \in \mathcal{X}$ must be equipped with a prime $p \in P$, written $x : p$. A control is used to construct complex terms. Each control $K \in \mathcal{K}$ must be equipped with an *arity rule* of the form

$$\frac{a_1 : m_1 \to n_1 \ \ldots \ a_r : m_r \to n_r}{K(a_1, \ldots, a_r) : m \to n}(\chi)$$

where χ may constrain the value of the integer r and the arities m_i, n_i, m, n. When fixed, r is called the *rank* of K. To define the reaction, we need a set R of *reaction rules*. A reaction rule is an ordered pair of terms having the same arity. The equational theory is common to all the control structures. Besides the ssmc and control operators, every control structure contains a *datum* operator $\langle x \rangle : \epsilon \to p$ (where $x : p$), a *discard* operator $\omega_p : p \to \epsilon$, and an *abstractor* operator $\mathtt{ab}_x\, a : p \otimes m \to p \otimes n$ (where $x : p$ and $a : m \to n$). Sometimes we will omit the arity subscripts. We suppose the terms used in our definitions and results are well-formed, and all the equations are between terms with the same arity.

The **action terms** are constructed by the following grammar:

$$a \ ::= \ \langle x \rangle \ \mid \ \omega_p \ \mid \ \mathtt{id}_m \ \mid \ \mathtt{p}_{m,n} \ \mid \ a \cdot b \ \mid \ a \otimes b \ \mid \ \mathtt{ab}_x\, a \ \mid \ K(a_1 \ldots a_r)$$

The following derived operator is a new kind of abstractor

$$(x)a \stackrel{def}{=} \mathtt{ab}_x\, a \cdot (\omega_p \otimes \mathtt{id}_n) \qquad (x : p, a : m \to n)$$
$$(\tilde{x})a \stackrel{def}{=} (x_1) \ldots (x_n)a \qquad (\tilde{x} = x_1 \ldots x_n)$$

The **equational theory** of a control structure is the congruence upon the terms generated by the ssmc axioms:

$$a \cdot \mathtt{id}_n = a \qquad \mathtt{p}_{m,n} \cdot \mathtt{p}_{n,m} = \mathtt{id}_{m \otimes n}$$
$$\mathtt{id}_m \cdot a = a \qquad \mathtt{p}_{k \otimes m,n} = (\mathtt{id}_k \otimes \mathtt{p}_{m,n}) \cdot (\mathtt{p}_{k,n} \otimes \mathtt{id}_m)$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \mathtt{p}_{m,n} \cdot (b \otimes a) = (a \otimes b) \cdot \mathtt{p}_{k,l}$$
$$a \otimes \mathtt{id}_\epsilon = a = \mathtt{id}_\epsilon \otimes a \qquad (a \otimes b) \otimes c = a \otimes (b \otimes c)$$
$$\mathtt{id}_m \otimes \mathtt{id}_n = \mathtt{id}_{m \otimes n} \qquad (a \cdot b) \otimes (c \cdot d) = (a \otimes c) \cdot (b \otimes d)$$

together with the following axioms (where x and y are different in $\mathtt{ab}_x \langle y \rangle = \mathtt{id}_p \otimes \langle y \rangle$):

$$\mathtt{ab}_x \langle x \rangle \cdot (\omega_p \otimes \mathtt{id}_p) = \mathtt{id}_p \qquad \mathtt{ab}_x\, \mathtt{p}_{m,n} = \mathtt{id}_p \otimes \mathtt{p}_{m,n}$$
$$\mathtt{ab}_x \langle y \rangle = \mathtt{id}_p \otimes \langle y \rangle \qquad \mathtt{ab}_x\, (a \cdot b) = \mathtt{ab}_x\, a \cdot \mathtt{ab}_x\, b$$
$$\mathtt{ab}_x\, \omega_q = \mathtt{id}_p \otimes \omega_q \qquad \mathtt{ab}_x\, (a \otimes \mathtt{id}_m) = \mathtt{ab}_x\, a \otimes \mathtt{id}_m$$
$$\mathtt{ab}_x\, \mathtt{id}_m = \mathtt{id}_{p \otimes m} \qquad \mathtt{ab}_x\, \mathtt{ab}_x\, a = \mathtt{id}_p \otimes \mathtt{ab}_x\, a$$
$$\mathtt{ab}_x\, \mathtt{ab}_y\, a \cdot (\mathtt{p}_{p,q} \otimes \mathtt{id}_n) = (\mathtt{p}_{p,q} \otimes \mathtt{id}_m) \cdot \mathtt{ab}_y\, \mathtt{ab}_x\, a \qquad (x : p, y : q, y \neq x)$$
$$(\langle x \rangle \otimes \mathtt{id}_m) \cdot (x)a = a \qquad (x : p, a : m \to n)$$
$$\langle x \rangle \cdot (y)(\langle y \rangle \otimes \langle y \rangle) = \langle x \rangle \otimes \langle x \rangle \qquad (x, y : p)$$
$$(\langle x \rangle \otimes \mathtt{id}_m) \cdot (y)K(a_1, \ldots) = K((\langle x \rangle \otimes \mathtt{id}_{m_1}) \cdot (y)a_1, \ldots) \qquad (x, y : p)$$

We use the equality $=$ between two actions a and b if the equation $a = b$ can be proved using the above axioms together with the rules of a congruence. Otherwise, we write $a \neq b$.

Each action a possesses a **surface** defined by $\mathtt{surf}(a) = \{x \in X \mid \mathtt{ab}_x\, a \neq \mathtt{id} \otimes a\}$.

We introduce a second derived operator by $[x/y]a \stackrel{def}{=} (\langle x \rangle \otimes \mathtt{id}_m) \cdot (y)a$. Two technical results motivate the substitution-like notation chosen for this derived operator. The first result shows a kind of α-conversion, and the second result shows properties verified by any standard capture-avoiding substitution (with the surface understood as the set of the free names). Note that, using this new derived operator, the last tree axioms of the equational theory become well-known substitution properties.

The **reaction** $\searrow$ is defined as the smallest relation upon the actions which satisfies the reaction rules R, and is closed under composition, tensor, abstraction, and equality.

3.2 Hypergraphs

In this section we recall the definition of the hypergraphs together with some standard related concepts such as isomorphism, contraction on nodes and edges. A **rooted hypergraph** is a tuple $H = \langle S, V, E, s \rangle$ where S is a set of *hyperedges*, V is a set of *vertices*, $E \subseteq S \times V$ is an *incidence* relation, and $s \in S$ is the *root hyperedge*. The components of a hypergraph H are denoted by S_H, V_H, E_H, and s_H. For a hyperedge $t \in S_H$ and a vertex $v \in V_H$, if $(t, v) \in E_H$ then we say that "v lies on t". The graphical representation of a hypergraph H is as follows. Hyperedges $t \in S_H$ are represented as unfilled ovals with the

name t outside. Vertices $v \in V_H$ are represented as points carrying the name v. Incidences $(t, v) \in E_H$ are represented as tentacles from the oval t to the point v; the length of such a tentacle is often taken to be zero. Finally, the root is emphasized by an arrow pointing to the oval s_H.

Example: Fig. 1 shows the graphical representation of a hypergraph H. Here $S_H = \{s, t, t'\}$, $V_H = \{v, w, w'\}$, $E_H = \{(s, v), (t, v), (t, w), (t', v), (t', w')\}$, and $s_H = \{t\}$. The left picture represents H using only tentacles of nonzero length, and the right one only tentacles of zero length. Often we use the representations of the middle picture as a compromise between size and clarity of visualization.

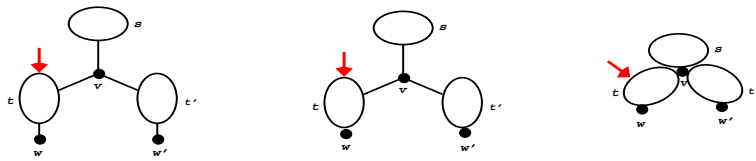


Fig. 1. Example of a hypergraph.

Let H be a rooted hypergraph. A nonempty subset of nodes $W \subseteq V_H$ gives rise to a **contraction on vertices** H/W which is the rooted hypergraph defined by

$S_{H/W} = S_H$
$V_{H/W} = (V_H \setminus W) \cup \{v\}$, for a fresh $v \notin V_H$
$E_{H/W} = (E_H \setminus S_H \times W) \cup \{\ (t, v) \mid \{t\} \times W \cap E_H \neq \emptyset\ \}$
$s_{H/W} = s_H$

A nonempty subset of hyperedges $T \subseteq S_H$ gives rise to a **contraction on hyperedges** H/T which is the rooted hypergraph defined by

$S_{H/T} = (S_H \setminus T) \cup \{t\}$, for a fresh $t \notin S_H$
$V_{H/T} = V_H$
$E_{H/T} = (E_H \setminus T \times V_H) \cup \{\ (t, v) \mid T \times \{v\} \cap E_H \neq \emptyset\ \}$
$s_{H/T} =$ `if` $s_H \in T$ `then` t `else` s_H

If we consider a hypergraph H, two vertices $v, w \in V_H$, and two hyperedges $s, t \in S_H$, then we write $H_{v=w}$ for the contraction on vertices $H/\{v, w\}$, and $H_{s=t}$ for the contraction on hyperedges $H/\{s, t\}$.

Example: Considering H the hypergraph used in the previous example, Fig. 2 shows a contraction $H_{w=w'}$ on vertices and a contraction $H_{t=t'}$ on hyperedges.

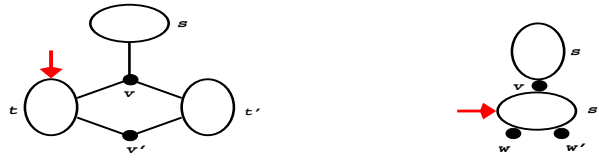


Fig. 2. Example of contractions on vertices and hyperedges.

Two hypergraphs H and H' are **isomorphic** if there exist two bijective functions $\phi_S : S_H \to S_{H'}$ and $\phi_V : V_H \to V_{H'}$ satisfying $\phi_S(s_H) = s_{H'}$ together with $(s, v) \in E_H$ if and only if $(\phi_S(s), \phi_V(v)) \in E_{H'}$ for all $s \in S_H$ and $v \in V_H$.

The relation of isomorphism is an equivalence over hypergraphs. For the present purpose, the names of hyperedges and vertices have no significance and we do not distinguish between isomorphic hypergraphs. We work with the equivalence classes of hypergraphs and we write $H = H'$ whenever H and H' are isomorphic. For the equivalence classes of hypergraphs we use the same graphical representation as for hypergraphs, removing the names of vertices and hyperedges.

3.3 The jc-Nets

In this subsection we define the control structure of jc-nets and provide a graphical representation for these nets. The arity monoid of this control structure is considered to be the additive monoid $(\mathbf{N}, +, 0)$ of the natural numbers. The symbols $m, n, k, \ldots$ range over natural numbers. $[n]$ denotes the first n naturals, i.e. $[n] = \{1, 2, \ldots, n\}$. Given a function $f : [n] \to Y$ and a natural k, we denote by $k \oplus f : \{k+1, \ldots k+n\} \to Y$ the function defined by $(k \oplus f)(i) = f(i-k)$. The control structure is defined over the set $\mathcal{X} = \{z_i \mid i \in \mathbf{N}\}$ of the join calculus names. The symbols $x, y, u, \ldots$ range over arbitrary names. The unique prime arity 1 is associated with each name $x \in \mathcal{X}$.

The actions of the control structure are enriched hypergraphs. An action $a = (H, \Sigma)$ of arity $m \to n$ consists of a hypergraph H together with a decoration $\Sigma = \langle \mathtt{I}, \mathtt{O}, \lambda, \tau, \mu \rangle$ of H consisting in an injective function $\mathtt{I} : [m] \to V_H$, a function $\mathtt{O} : [n] \to V_H$, an injective function $\lambda : Z \to V_H$ with $Z \subseteq \mathcal{X}$, a relation $\tau \subseteq V_H \times V_H$, and a function $\mu : S_H \to \mathbf{N}^{V_H \times V_H}$. The functions μ can be viewed as multisets over $S_H \times V_H \times V_H$. We use multiset operations $\uplus, -, \ldots$ over these functions. We use the notation $\{x, y, y\}$ for an arbitrary multiset μ over the set $\{x, y, z\}$ with $\mu(x) = 1$, $\mu(y) = 2$ and $\mu(z) = 0$.

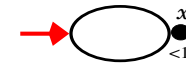
The concepts of isomorphism and contraction introduced for hypergraphs extend in a quite straightforward way to nets. Let $a_i = (H_i, \Sigma_i)$ with $\Sigma_i = \langle \mathtt{I}_i, \mathtt{O}_i, \lambda_i, \tau_i, \mu_i \rangle$, where $i \in [2]$. We say that a_1 and a_2 are isomorphic if there exists a hypergraph isomorphism (ϕ_S, ϕ_V) between H_1 and H_2 such that $\phi_V \circ \mathtt{I}_1 = \mathtt{I}_2$, $\phi_V \circ \mathtt{O}_1 = \mathtt{O}_2$, $\phi_V \circ \lambda_1 = \lambda_2$, together with $(v, v') \in \tau_1$ if and only if $(\phi_V(v), \phi_V(v')) \in \tau_2$ and $\mu_1(s, v, v') = \mu_2(\phi_S(s), \phi_V(v), \phi_V(v'))$ for all $s \in S_{H_1}$ and $v, v' \in V_{H_1}$. Just as for hypergraphs, we do not distinguish between isomorphic nets.

We now explain the graphical representations of nets. Let $a = (H, \Sigma)$ be a net with $\Sigma = \langle \mathtt{I}, \mathtt{O}, \lambda, \tau, \mu \rangle$. First we represent the hypergraph H as explained in the previous subsection. We suppose that the tentacles in H are of length zero. If $\mathtt{I}(i) = v$, $\mathtt{O}(k) = v'$ and $\lambda(x) = w$, then we assign an *input label* (i) to the vertex v, an *output label* $\langle k \rangle$ to the vertex v', and a *name label* x to the vertex w. If $(v, v') \in \tau$, then we draw an arc outside any oval from the vertex v to the vertex v'. It is easy to show that for jc-nets, if $\mu(s, v, v') > 0$, then v and v' lie on the same hyperarc s, i.e. $(s, v), (s, v') \in E_H$. Thus, if $\mu(s, v, v') = k > 0$, we can draw k arcs inside the oval s from the vertex v to the vertex v'. Similar to hypergraphs, for the isomorphism classes of nets we use the same graphical representation as for nets; the only thing is to eliminate the names of vertices and of hyperedges.

We introduce now the control structure operators of the jc-nets.
The **datum** $\langle x \rangle^\gamma = (H, \Sigma) : 0 \to 1$ is defined by

$H = \langle \{s\}, \{v\}, \{(s, v)\}, s \rangle$

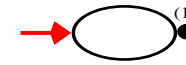
$\Sigma = \langle \emptyset, \{1 \mapsto v\}, \{x \mapsto v\}, \emptyset, \emptyset \rangle$



The **discard** $\omega^\gamma = (H, \Sigma) : 1 \to 0$ is defined by

$H = \langle \{s\}, \{v\}, \{(s, v)\}, s \rangle$

$\Sigma = \langle \{1 \mapsto v\}, \emptyset, \emptyset, \emptyset, \emptyset \rangle$

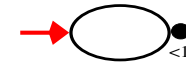


The three **controls** which generate the jc-nets are:

– $\nu^\gamma = (H, \Sigma) : 0 \to 1$ is defined by

$H = \langle \{s\}, \{v\}, \{(s, v)\}, s \rangle$

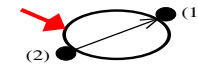
$\Sigma = \langle \emptyset, \{1 \mapsto v\}, \emptyset, \emptyset, \emptyset \rangle$



– $\mathtt{out}^\gamma = (H, \Sigma) : 2 \to 0$ is defined by

$H = \langle \{s\}, \{v, v'\}, \{(s, v), (s, v')\}, s \rangle$

$\Sigma = \langle \emptyset, \{1 \mapsto v, 2 \mapsto v'\}, \emptyset, \emptyset, \{(s, v', v)\} \rangle$



– If $a = (H, \Sigma) : 1 \to 0$, $\Sigma = \langle \mathtt{I}, \mathtt{O}, \lambda, \tau, \mu \rangle$ then $\mathtt{def}^\gamma\, a = (H', \Sigma') : 1 \to 0$ with

$H' = \langle S_H \cup \{t\}, V_H \cup \{v\}, E_H \cup \{(t, v)\}, t \rangle$, for fresh $t \notin S_H$ and $v \notin V_H$

$\Sigma' = \langle \{1 \mapsto v\}, \mathtt{O}, \lambda, \tau \cup \{(v, \mathtt{I}(1))\}, \mu \rangle$

The **ssmc operators** are defined in the following way. Consider $a_i = (H_i, \Sigma_i)$ with $\Sigma_i = \langle \mathtt{I}_i, \mathtt{O}_i, \lambda_i, \tau_i, \mu_i \rangle$ and $\lambda_i : Z_i \to V_{H_i}$, where $i \in [2]$. W.l.o.g. we suppose $s_{H_1} = s_{H_2} = s$ and $(S_{H_1} - \{s_{H_1}\}) \cap (S_{H_2} - \{s_{H_2}\}) = \emptyset$, as well as $\lambda_1(z) = \lambda_2(z)$, $\forall z \in Z_1 \cap Z_2$ and $(V_{H_1} - \lambda_1(Z_1 \cap Z_2)) \cap (V_{H_2} - \lambda_2(Z_1 \cap Z_2)) = \emptyset$.

– *Identity* $\mathtt{id}^\gamma_m = (H, \Sigma) : m \to m$ is defined by

$H = \langle \{s\}, \{v_i | i \in [m]\}, \{(s, v_i) | i \in [m]\}, s \rangle$

$\Sigma = \langle \{i \mapsto v_i | i \in [m]\}, \{i \mapsto v_i | i \in [m]\}, \emptyset, \emptyset, \emptyset \rangle$

– *Symmetry* $\mathtt{p}^\gamma_{m,n} = (H, \Sigma) : m + n \to n + m$ is defined by

$H = \langle \{s\}, \{v_i | i \in [m + n]\}, \{(s, v_i) | i \in [m + n]\}, s \rangle$

$\Sigma = \langle \{i \mapsto v_i | i \in [m + n]\}, \{i \mapsto v_{m+i} | i \in [n]\} \cup \{n + i \mapsto v_i | i \in [m]\}, \emptyset, \emptyset, \emptyset \rangle$

– *Tensorial product* $a_1 \otimes a_2 : m + k \to n + l$ of $a_1 : m \to n$ and $a_2 : k \to l$ is obtained by combining a_1 and a_2 as follows. Increment with m the input labels, and with n the output labels in a_2. Contract the two roots, as well as vertices of a_1 and of a_2 bearing the same name label. Formally, $a_1 \otimes a_2 = (H, \Sigma)$ where

$H = \langle S_{H_1} \cup S_{H_2}, V_{H_1} \cup V_{H_2}, E_{H_1} \cup E_{H_2}, s \rangle$

$\Sigma = \langle \{\mathtt{I}_1 \cup m \oplus \mathtt{I}_2, \mathtt{O}_1 \cup n \oplus \mathtt{O}_2, \lambda_1 \cup \lambda_2, \tau_1 \cup \tau_2, \mu_1 \uplus \mu_2 \rangle$

– *Composition* $a_1 \cdot a_2 : m \to k$ of $a_1 : m \to n$ and $a_2 : n \to k$ is obtained by combining a_1 and a_2 as follows. Contract the two roots, as well as vertices of a_1 and of a_2 bearing the same name label. For each $i \in [n]$, contract the vertex of a_1 labeled by $\langle i \rangle$ with the vertex of a_2 labeled by (i). Remove the labels (i) and $\langle i \rangle$. Formally, $a_1 \cdot a_2 = (H, \Sigma)_{\mathtt{O}_1(1)=\mathtt{I}_2(1),\ldots,\mathtt{O}_1(n)=\mathtt{I}_2(n)}$ where

$H = \langle S_{H_1} \cup S_{H_2}, V_{H_1} \cup V_{H_2}, E_{H_1} \cup E_{H_2}, s \rangle$
$\Sigma = \langle \{\mathtt{I}_1, \mathtt{O}_2, \lambda_1 \cup \lambda_2, \tau_1 \cup \tau_2, \mu_1 \uplus \mu_2 \rangle$

We define the **abstractor**. Let $a = (H, \Sigma) : m \to n$ with $\Sigma = \langle \mathtt{I}, \mathtt{O}, \lambda, \tau, \mu \rangle$. Then $\mathtt{ab}_x^\gamma\, a : 1 + m \to 1 + n$ is obtained from a as follows. Increment with 1 all input and output labels. Assign to the vertex labeled by x both the input label (1) and the output label $\langle 1 \rangle$. Remove the label x. Formally, $\mathtt{ab}_x^\gamma\, a = (H, \Sigma')$ where $\Sigma' = \langle \{1 \mapsto \lambda(x)\} \cup 1 \oplus \mathtt{I}, \{1 \mapsto \lambda(x)\} \cup 1 \oplus \mathtt{O}, \lambda - \{x \mapsto \lambda(x)\}, \tau, \mu \rangle$.

It is easy to see that the above introduced operators over nets are well-defined, except the abstractor. Indeed, $\mathtt{ab}_x^\gamma\, a$ is not well-defined if the net a does not contain a vertex labeled by x. We need an additional technical adjustment. Thus we change the definition of the above introduced operators by $\mathtt{op}(a, \ldots) \stackrel{def}{=} \mathtt{op}^\gamma(a \otimes^\gamma \mathbf{i}, \ldots) \otimes^\gamma \mathbf{i}$, where $\mathtt{op}$ stands for each of these operators and $\mathbf{i} = (H, \Sigma)$ is the net defined by

$H = \langle \{s\}, \{v_i | i \in \mathbf{N}\}, \{(s, v_i) | i \in \mathbf{N}\}, s \rangle$
$\Sigma = \langle \emptyset, \emptyset, \{z_i \mapsto v_i | i \in \mathbf{N}\}, \emptyset, \emptyset \rangle$.

Proposition 1. *The operators* $\langle x \rangle$, ω, ν, $\mathtt{out}$, $\mathtt{def}$, $\mathtt{id}$, $\mathtt{p}$, $\cdot$, $\otimes$ *and* $\mathtt{ab}_x$ *define a control structure.*

We define the following two derived control operators:

$$\mathtt{out}_u \stackrel{def}{=} (\langle u \rangle \otimes \mathtt{id}_1) \cdot \mathtt{out}$$
$$\mathtt{def}_u\, a \stackrel{def}{=} \langle u \rangle \cdot \mathtt{def}\, a$$

The operator $\mathtt{def}$ can be generalized. If $a : m \to 0$, then $\mathtt{def}\, a : m \to 0$. The corresponding graphical representation is extended in a natural way (we use m external arcs to connect the new root hyperarc to the old one). We can define a derived control operator $\mathtt{def}_{u_1 \ldots u_m}$ by

$$\mathtt{def}_{u_1 \ldots u_m}\, a = (\langle u_1 \rangle \otimes \ldots \otimes \langle u_m \rangle) \cdot \mathtt{def}\, a$$

The **reaction** $\searrow$ of the control structure is the smallest relation over $\mathtt{jc}$-nets closed under tensorial product, composition, abstraction, and equality, which satisfies the following control rule

$$\mathtt{out}_u \otimes \mathtt{def}_u\, a \searrow a \otimes \mathtt{def}_u\, a$$

The reaction can be generalized in the following way

$$\mathtt{out}_{u_1} \otimes \ldots \otimes \mathtt{out}_{u_m} \otimes \mathtt{def}_{u_1 \ldots u_m}\, a \searrow a \otimes \mathtt{def}_{u_1 \ldots u_m}\, a$$

4 Expressiveness of the jc-Nets

We give the semantics of the join calculus by using our $\mathtt{jc}$-nets. It is known that the join calculus has the same expressive power as the π-calculus [Fou99]. The results of this section show that our $\mathtt{jc}$-nets have the same expressive power as the join calculus.

Definition 4. *The encoding of the join calculus processes using the* $\mathtt{jc}$*-nets is defined by*

1. $[0] = \mathtt{id}_0$
2. $[u\langle v\rangle] = \langle v\rangle \cdot \mathtt{out}_u$
3. $[P \mid Q] = [P] \otimes [Q]$
4. $[\mathtt{def}\ u\langle y\rangle \triangleright P\ \mathtt{in}\ Q] = \nu \cdot (u)([Q] \otimes \mathtt{def}_u\,(y)[P])$

Lemma 1. $\mathtt{fn}(P) \supseteq \mathtt{surf}([P])$.

Lemma 2. $[\{x/y\}P] = [x/y][P]$

Proposition 2. *If $P \equiv Q$, then we have $[P] = [Q]$.*

Theorem 1. *If $P \to Q$, then we have $[P] \searrow [Q]$.*

Theorem 2. *If $[P] \searrow a$, then there is a process Q such that $P \to Q$ and $[Q] = a$.*

The encoding of the full join calculus by the `jc`-nets can be easily obtained in the monadic version because the number of the defined channels is equal to the number of received channels for each join-pattern. If $\mathtt{dn}(D) = \{u_1, \ldots, u_k\}$ and $J = u_1\langle y_1\rangle | \ldots | u_n\langle y_n\rangle$, then we define the semantics of the full monadic join calculus by:

$$[0] = \mathtt{id}_0 \qquad [u\langle v\rangle] = \langle v\rangle \cdot \mathtt{out}_u \qquad [D \wedge E] = [D] \otimes [E]$$
$$[P \mid Q] = [P] \otimes [Q] \qquad [J \triangleright P] = \mathtt{def}_{u_1 \ldots u_n}\,(y_1 \ldots y_n)[P]$$
$$[\mathtt{def}\ D\ \mathtt{in}\ P] = (\nu \otimes \ldots \otimes \nu) \cdot (u_1 \ldots u_k)([D] \otimes [P])$$

It is not necessary to encode the join-patterns because the join-patterns interfere just by the defined and received channels.

5 Example: A Chat System

In this section we implement a simple chat system using our `jc`-nets. A chat system consists of a server and several clients and it is defined by the following property: each message coming at the input of any client must be present at the output of all clients. Similar to the mechanism of Unix message queues, we use a single channel q to exchange messages between the server and the clients. A type is associated with each message to allow the server and the clients to multiplex messages onto an unique q. Let us consider a chat system made up of a server S and two clients A and B. A channel `idS` is used to provide the type of a message from any client to the server. A channel `idA` (respectively `idB`) is used to provide the type of a message from the server to the client A (respectively B). The client A (respectively B) uses an input channel `inA` and an output channel **outA** (respectively `inB` and **outB**). Considering that a message `m` is sent along the input channel `inA`, the join calculus program corresponding to this chat system is as follows:

$$\begin{aligned} Chat = \mathtt{def}\ & q\langle x\rangle \mid idS\langle y\rangle \ \triangleright\ q\langle x\rangle \mid idA\langle\_\rangle \mid q\langle x\rangle \mid idB\langle\_\rangle \ \wedge \\ & q\langle x\rangle \mid idA\langle y\rangle \ \triangleright\ outA\langle x\rangle \ \wedge \\ & q\langle x\rangle \mid idB\langle y\rangle \ \triangleright\ outB\langle x\rangle \ \wedge \\ & inA\langle x\rangle \ \triangleright\ q\langle x\rangle \mid idS\langle\_\rangle \ \wedge \\ & inB\langle x\rangle \ \triangleright\ q\langle x\rangle \mid idS\langle\_\rangle \\ \mathtt{in}\ & inA\langle m\rangle \end{aligned}$$

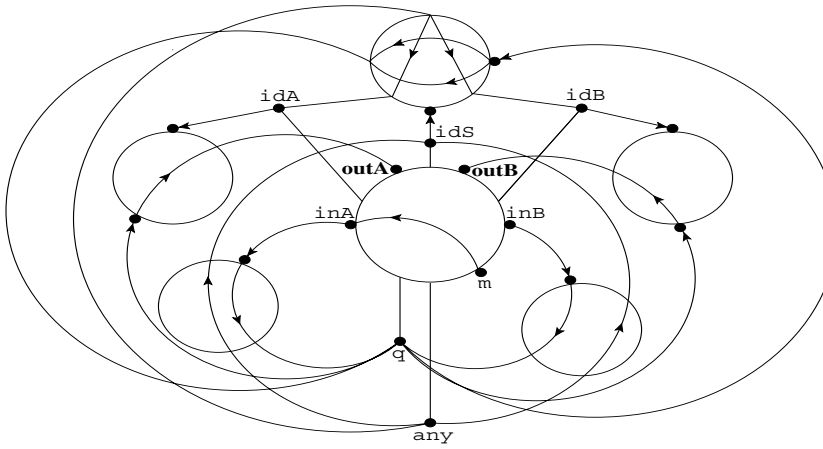


Fig. 3. The jc-net describing the chat system.

According to the semantics provided, the jc-net of Fig. 3 corresponds to the above join calculus program. Despite the fact that the only proper labels of the jc-net are **outA** and **outB**, we use other labels as well in order to give a clear description of the chat system. The structure of the described jc-net is not changing[1] during the reaction steps. Only the root hyperarc is changing. Thus we can visualize the reactions of the described jc-net focusing just on the changes of the root hyperarc.

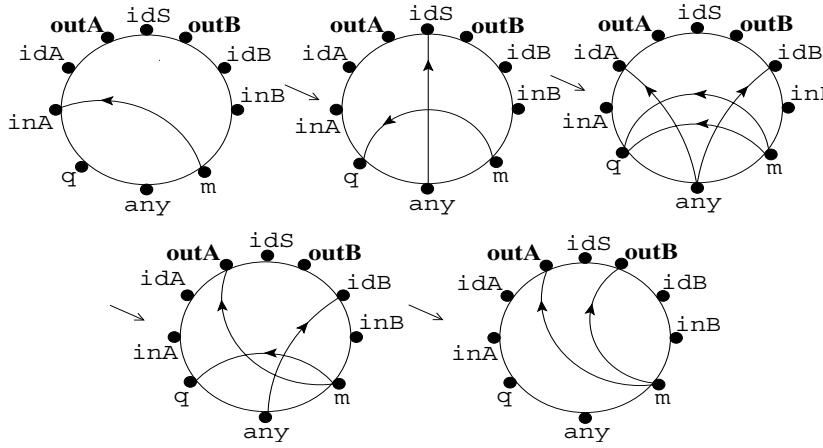


Fig. 4. The reactions of the jc-net describing the chat system.

[1] Generally, the structure of a jc-net may expand and shrink during reaction, namely new hyperarcs may appear by hyperarc duplication or existing hyperarcs can disappear by hyperarc fusion.

It is easy to see in Fig. 4 that the described system satisfies the chat condition: the message `m` coming along the input channel `inA` in the initial `jc`-net is sent to the output channels **outA** and **outB** in the final `jc`-net.

6 Conclusion and Related Works

We have introduced and studied a graphical representation for the join calculus given by the `jc`-nets. Why is interesting this new graphical formalism? The `jc`-nets avoid irrelevant aspects of π-nets without losing their expressive power. Moreover, our formalism is presented by the same algebraic framework used for the presentation of the π-nets.

Graphical representations for the π-calculus or for other process calculi highlights new perceptions and provide more intuition about concurrency and distributed systems. While it has always been usual to draw processes as flow-graphs for a more intuitive representation, there are, as far as we know, only few papers formalizing the graphical presentations of processes. Two graphical formalisms were given by Robin Milner: π-nets and action graphs [Mil94,Mil96]. The action graphs are graphical representations for action calculi; being too general, they are not able to catch specific features of some particular action calculi. Gabriel Ciobanu and Mihai Rotaru have studied the faithful π-nets [CR98] and tree resource automata with their graphical representation [CR00]. Barbara König has developed a different hypergraph-based formalism for mobility, providing a hypergraph rewriting semantics for the π-calculus [Kon99].

References

CR98. G. Ciobanu, M. Rotaru. "Faithful π-nets. A Graphical Representation of the Asynchronous π-calculus", ENTCS, vol.18, 22 p, North-Holland, 1998.

CR00. G. Ciobanu, M. Rotaru. "A π-calculus Machine". In *Journal of Universal Computer Science*, vol.6(1), 39-59, Springer, 2000.

FG96. C. Fournet, G. Gonthier. "The Reflexive CHAM and the Join Calculus". In *Proc. POPL'96*, ACM Press, 1996.

Fou99. C. Fournet. "The Join-Calculus: A calculus for Distributed Mobile Programming", PhD thesis, INRIA Rocquencourt, 1998.

Kon99. B. König. "Description and Verification of Mobile Processes with Graph Rewriting Techniques", PhD thesis, Technische Universität München, 1999.

Lev98. J.J. Lévy. "Some Results in the Join Calculus". In *Proc.TACS'97*, LNCS vol.1281, Springer, 233-249, 1997.

MMP95. A. Mifsud, R. Milner, A.J. Parrow. "Control Structures". In *Proc. LICS'95*.

Mif96. A. Mifsud. PhD thesis, University of Edinburgh, 1996.

Mil93. R. Milner. "The Polyadic π-calculus: a Tutorial". In *Logic and Algebra of Specification*, Springer, 203-246, 1993.

Mil94. R. Milner. "π-nets: a Graphical Form of π-calculus". In *Proc. ESOP'94*, LNCS 788, Springer, 26-42, 1994.

Mil96. R. Milner. "Calculi for Interaction", Acta Informatica 33 (8), 707-737, 1996.

Mil99. R. Milner. *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999.

Nonterminal Complexity of Programmed Grammars

Henning Fernau

Wilhelm-Schickard-Institut für Informatik; Universität Tübingen
Sand 13; D-72076 Tübingen; Germany
fernau@informatik.uni-tuebingen.de
http://www-fs.informatik.uni-tuebingen.de/~fernau

Abstract. We show that, in the case of context-free programmed grammars with appearance checking working under free derivations, three nonterminals are enough to generate every recursively enumerable language. This improves the previously published bound of eight for the nonterminal complexity of these grammars. This also yields an improved nonterminal complexity bound of four for context-free matrix grammars with appearance checking. Moreover, we establish nonterminal complexity bounds for context-free programmed and matrix grammars working under leftmost derivations.

1 Introduction

Descriptional complexity (or, more specifically, syntactic complexity) is interested in measuring the complexity of describing objects (in our case, formal languages) with respect to different syntactic complexity measures. In particular, very economical presentations of languages are sought for. For example, Shannon [14] showed the nowadays classical result that every recursively enumerable language can be accepted by some Turing machine with only two states.

Similar complexity considerations may be carried out for any language describing device. In the case of grammars, natural syntactic complexity measures are the number of nonterminals and the number of rewriting rules. In this paper, we will consider the nonterminal complexity of certain regulated grammar formalisms which characterize the recursively enumerable languages. In the literature, several interesting results on this topic appeared in recent years. For example, in the case of scattered context grammars, there has even been some sort of race for the smallest possible complexity bound, see [8,9,10]. Here , we will concentrate on the question: how many nonterminals must a context-free programmed grammar (working under free derivation) with appearance checking necessarily have in order to be able to generate every recursively enumerable language? Previously, a solution using eight nonterminals has been known [1, Theorem 4.2.3]. We improve this bound to three by using a rather intricate Turing machine simulation. This is our main result. This result could also be useful within the emerging area of membrane computing [5]. In fact, Freund and Păun derived an upper bound of four for the nonterminal complexity of programmed

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 202–213, 2001.

grammars [5] by using a different simulation technique based on register machines. As a corollary, we derive that three nonterminals are enough to generate every recursively enumerable language by using context-free programmed grammars with appearance checking working under leftmost derivations of type 3. The same bound was previously claimed for context-free programmed grammars without appearance checking working under leftmost derivations of type 2 by Meduna and Horváth [11, Theorem 5] (within Kasai's formalism of state grammars [7]). Since we think that the proof given there is incorrect, we give a new characterization of the recursively enumerable languages through programmed grammars without appearance checking working under leftmost derivations of type 2 with four nonterminals based on the construction leading to our main theorem. Similarly, a nonterminal bound of four can be derived for grammars with unconditional transfer checking working under leftmost derivations of type 2. Our main result also yields an improved nonterminal complexity bound for context-free matrix grammars with appearance checking (namely four instead of six as previously published in [12], also see [1, Theorem 4.2.3]; independently, this bound was achieved recently by Freund and Păun [5]). This bound holds for matrix grammars working under free derivations and working under leftmost derivations of type 3, as well. Finally, we can derive a first nonterminal complexity bound for context-free programmed (or matrix) grammars with unconditional transfer (working under leftmost derivations of type 3), under the assumption that the terminal alphabet is fixed. A long version is available as Technical Report WSI–2000–26, Wilhelm-Schickard-Institut für Informatik, Tübingen.

We use standard mathematical and formal language notations throughout the paper, as they can be found in [1,6]. In particular: π_j selects the jth component of an n-tuple; λ denotes the empty word; w^R denotes the reversal of string w.

2 Turing Machines

In order to be able to reason more formally, in the following, we give a definition of a Turing machine which is adapted to our purposes. The reader can probably easily check its equivalence with his or her favourite definition.

Definition 1. *A* (nondeterministic) Turing machine (with one one-sided tape) *is given by* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f, \#_L, \#_R, \#)$, *where* Q *is the state alphabet,* Σ *is the input alphabet,* Γ *(with* $\Sigma \subseteq \Gamma$ *and* $\Gamma \cap Q = \emptyset$*) is the tape alphabet,* $\delta \subseteq Q \times \Gamma \times \{L, R\} \times Q \times \Gamma$ *is the transition relation,* q_0 *is the initial state,* q_f *is the final state,* $\#_L \in \Gamma$ *is the left endmarker,* $\#_R \in \Gamma$ *is the right endmarker and* $\# \in \Gamma$ *is the blank symbol.*

A configuration *(also called instantaneous description) of* M *is described by a word* $c \in \#_L\tilde{\Gamma}^*\#_R Q \cup \#_L\tilde{\Gamma}^*Q\tilde{\Gamma}^*\#_R$ *with* $\tilde{\Gamma} = \Gamma \setminus \{\#_L, \#_R\}$. *Here* $c = \#_L wqv$ *means: The head of the Turing machine is currently scanning the last symbol* a *of* $\#_L w$. *We now describe possible (and the only possible) successor configurations* c' *of* $c = \#_L wqv$ *(given* δ*), written* $c \vdash_M c'$ *for short:*

1. *If* $a \neq \#_L$, $a \neq \#_R$ *and* $(q, a, L, q', a') \in \delta$ *with* $a' \in \tilde{\Gamma}$, *then for* $w = w'a$, $c = \#_L w'aqv \vdash_M \#_L w'q'a'v$ *holds.*

2. *If* $a \neq \#_L$, $a \neq \#_R$ *and* $(q, a, R, q', a') \in \delta$ *with* $a' \in \tilde{\Gamma}$, *then for* $w = w'a$ *and* $v = bv'$ *with* $b \in \Gamma \setminus \{\#_L\}$, $c = \#_L w'aqbv' \vdash_M \#_L wbq'v'$ *holds.*
3. *If* $a = \#_L$ *and* $(q, \#_L, R, q', \#_L) \in \delta$, *then for* $w = \lambda$ *and* $v = bv'$ *with* $b \in \Gamma \setminus \{\#_L\}$, $c = \#_L qbv' \vdash_M \#_L bq'v'$ *holds.*
4. *If* $a = \#_R$ *and* $(q, \#_R, L, q', a') \in \delta$ *with* $a' \in \tilde{\Gamma}$, *then for* $w = w'\#_R$, $c = \#_L w'\#_R q \vdash_M \#_L w'q'a'\#_R$ *holds.*

As usual, $\vdash_M^*$ *denotes the reflexive transitive hull of the binary relation* $\vdash_M$. *The language* generated *by* M *is given as:*

$$L(M) = \{w \in \Sigma^* \mid \#_L q_0 \#_R \vdash_M^* \#_L w \xi q_f \#_R, \text{ where } \xi \in \#^*\}.$$

Observe that only the last condition given in the definition of successor configuration allows for prolongating the working tape. This ability is, of course, essential for obtaining the power to describe all recursively enumerable languages, see also the famous workspace theorem [6]. In the following, $\mathcal{RE}$ denotes the class of all recursively enumerable languages, which can be characterized as the family of languages generatable by Turing machines.

3 Regulated Grammars

The notion of a programmed grammar is crucial to this paper.

Definition 2. *A* (context-free) programmed grammar (with appearance checking) *is given by a quadruple* $G = (N, \Sigma, P, S)$, *where* N *is the nonterminal alphabet,* Σ *is the terminal alphabet,* $S \in N$ *is the start symbol and* P *is a finite set of rules of the form* $(r : A \to w, \sigma(r), \phi(r))$, *where* $r : A \to w$ *is a context-free rewriting rule, i.e.,* $A \in N$ *and* $w \in (N \cup \Sigma)^*$ *(hence, erasing rules are permitted), which is labelled by* r, *and* $\sigma(r)$ *and* $\phi(r)$ *are two sets of labels of such context-free rules appearing in* P. $A \to w$ *is termed* core rule *of* $(r : A \to w, \sigma(r), \phi(r))$. $\sigma(r)$ *is also called* success field *of* r *and* $\phi(r)$ *is called* failure field *of* r. *By* $\Lambda(P)$, *we denote the set of all labels of the rules appearing in* P.

For $(x_1, r_1), (x_2, r_2) \in (N \cup \Sigma)^* \times \Lambda(P)$, *we write* $(x_1, r_1) \Rightarrow (x_2, r_2)$ *iff either*

$$x_1 = yAz,\ x_2 = ywz,\ (r_1 : A \to w, \sigma(r_1), \phi(r_1)) \in P, \text{ and } r_2 \in \sigma(r_1)$$

or $x_1 = x_2$, $(r_1 : A \to w, \sigma(r_1), \phi(r_1)) \in P$, A *does not occur in* x_1 *and* $r_2 \in \phi(r_1)$. *Let* $\stackrel{*}{\Rightarrow}$ *denote the reflexive transitive hull of* $\Rightarrow$. *The language generated by* G *is defined as* $L(G) = \{w \in \Sigma^* \mid (S, r_1) \stackrel{*}{\Rightarrow} (w, r_2)$ *for some* $r_1, r_2 \in \Lambda(P)\}$. *The language family generated by programmed grammars is denoted by* $\mathcal{P}$. *The language family generated by programmed grammars with at most* k *nonterminals is denoted by* $\mathcal{P}_k$.[1]

[1] In any case, a number as subscript in the corresponding language class denotation will refer to a nonterminal bound for that class.

Theorem 1. $\mathcal{RE} = \mathcal{P}_8 = \mathcal{P}$. *(see [1])*

Dassow and Păun pose as an open question whether or not that complexity bound could be improved. We address this question here.

In the literature, two variants of programmed grammars are discussed:

- In a programmed grammar $G = (N, \Sigma, P, S)$ *without appearance checking*, for every $r \in \Lambda(P)$, we have $\phi(r) = \emptyset$.
- In a programmed grammar $G = (N, \Sigma, P, S)$ *with unconditional transfer*, for every $r \in \Lambda(P)$, we have $\phi(r) = \sigma(r)$.

In several papers, *leftmost* derivations are introduced, as well. For precise definitions of leftmost derivations of types 3 and 2, we refer to [1,3]. For better distinguishability from leftmost derivations, we will call the derivation relation defined above *free derivation*. Let $\mathcal{P}^{\ell-3}$ denote the language family generatable by context-free programmed grammars working under *leftmost derivation of type 3*, i.e., a selected rule $(r : A \to w, \sigma(r), \phi(r))$ of a context-free programmed grammar is applied to a sentential form α always in a manner choosing the leftmost occurrence of A in α for replacement. Let $\mathcal{PUT}^{\ell-3}$ denote the class of languages generatable by context-free programmed grammars with unconditional transfer working under leftmost derivations of type 3. Let $\mathcal{P}^{\ell-2}$ denote the language family generatable by context-free programmed grammars without appearance checking working under *leftmost derivation of type 2*. Such a grammar is also known as *state grammar* [7,11]. Let $\mathcal{PUT}^{\ell-2}$ denote the class of languages generatable by context-free programmed grammars with unconditional transfer working under leftmost derivations of type 2.

Theorem 2. $\mathcal{RE} = \mathcal{P}^{\ell-3} = \mathcal{PUT}^{\ell-3} = \mathcal{P}^{\ell-2} = \mathcal{PUT}^{\ell-2}$. *(see [1,4,13])*

For reasons of space, we do not define context-free matrix grammars formally, but refer the reader to [1]. The language families are denoted by replacing the $\mathcal{P}$ in the programmed language family denotation with $\mathcal{M}$.

Theorem 3. $\mathcal{RE} = \mathcal{M}_6 = \mathcal{M} = \mathcal{M}^{\ell-3} = \mathcal{MUT}^{\ell-3}$. *(see [1,3,4])*

4 Main Result

In this section, we are going to sketch a proof of the following result, thereby improving the previously known nonterminal complexity bound considerably:

Theorem 4. $\mathcal{RE} = \mathcal{P}_3$.

Due to Theorem 1, only the inclusion $\subseteq$ has to be shown.

4.1 Informal Explanations

We proceed by giving several explanations concerning our construction on a rather intuitive level. Of course, we only need to show how to simulate a Turing machine generating some language L by a programmed grammar with three nonterminals. The three nonterminals of the simulating grammar are: A, B, and C. We consider a fixed Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f, \#_L, \#_R, \#)$.

Encodings. Given a configuration $c \in (\Gamma \cup Q)^*$ of M, let $\beta(c) \in \{0,1\}^*$ denote some binary encoding of c using $\gamma = \lceil \log_2(|\Gamma| + |Q|) \rceil$ many bits per symbol from $\Gamma \cup Q$. Since Γ contains at least three special symbols, namely, $\#_L$, $\#_R$ and $\#$, and one input symbol and since Q contains at least one symbol, we have $\gamma \geq 3$. We interpret strings $\beta(c)$ as natural numbers given in binary in a standard manner, i.e., 001 for example, would be the three-bit binary representation of the number 1. We further assume $\beta(\#) = 0^\gamma$, $\beta(\#_L) = 0^{\gamma-1}1$ and $\beta(\#_R) = 10^{\gamma-1}$. Observe that $|\beta(c)| = |c|\gamma$. Obviously, c can be codified uniquely over the unary alphabet $\{A\}$ by $A^{\beta(c)}$, which is not the empty word, because c always starts with $\#_L$.

There is a special technique we call "passing over symbols" which we describe next.

Passing over Symbols. How can a tape symbol be "passed over", e.g., in the simulation phase? To this end, consider the following program fragment:

$$\begin{array}{l}((p,1) : A \to \lambda,\ \{(p,2)\}, \{(p,3)\}) \\ ((p,2) : A \to C,\ \{(p,1)\}, \{(p,3)\}) \\ ((p,3) : C \to A,\ \{(p,3)\}, \text{exit})\end{array}$$

Such a program fragment is useful to transfer A^n into A^m with $m = \lfloor n/2 \rfloor$. If $n = \beta(c)$, then the *rightmost* bit of $\beta(c)$ is erased. Such a loop is useful in several circumstances:

1. within the simulation loop, in order to pass over symbols which are uninteresting in the simulated step; here, one has to store the skipped symbols somehow (to this end, the symbol B is going to be used);
2. when checking the correctness of the guess on where to actually start the simulation of one Turing machine step;
3. when transforming a codified terminal string, e.g., $A^{\beta(\#_L w q_f \#_R)}$, into w with $w \in \Sigma^*$.

In each of the three described situations, the two different branches towards $(p,3)$ could be used to test the contents of the currently last bit of the string x stored in A^x.

Simulation Loop. Our aim is to give several rules of the programmed grammar such that $c \vdash_M c'$ is reflected by $(A^{\beta(c)}B, p) \stackrel{*}{\Rightarrow} (A^{\beta(c')}B, p')$. We assume that the Turing machine state q of configuration c is somehow stored in the label p.

The grammar scans $A^{\beta(c)}$, searching for some codification of the subword aq, where the assumed input symbol a is guessed nondeterministically. To this end, a nondeterministically chosen number of letters is passed over, until the guessed subword aq is chosen to be verified. At the end of the verification of the correctness of the subword guess (at the chosen position), a replacement (chosen from the possibilities given by δ, hence yielding c' from c) is simulated. Finally, the intermediate representation has to be converted back into the standard codification $A^{\beta(c')}$.

4.2 A Formal Construction

Initialization. Consider

$$(\text{init} : A \to A^{\beta(\#_L q_0 \#_R)} B, \text{simstart}(q_0), \emptyset)$$

as the start rule. Here, simstart is a label set indicating possible starting points of the simulation. simstart will be defined formally below.

Remark 1. There are two intricate differences between programmed and graph-controlled grammars: (1) No distinguished start label and (2) no distinguished final label are specified in programmed grammars (in contrast to graph-controlled grammars). In order to cope with these shortcomings, our simulating programmed grammar is designed in a way that the axiom A never directly derives a terminal string. Instead, e.g., $A \to \lambda$ (as occurring in the explanatory subsection 4.1) is replaced by three rules in a sequence: (i) $A \to C$, (ii) $B \to B$ (checking for the presence of B) and (iii) $C \to \lambda$. This sequence can be successfully applied only to sentential forms containing a B. This means that init is the only possible starting point.

Remark 2. One could have avoided such additional complication by considering graph-controlled grammars as suggested in [2] as a possible clearer grammatical model. In fact, all results of this paper as stated for programmed grammars are also valid for this related grammatical mechanism. Freund and Păun independently proved [5] that three nonterminals are enough for characterizing $\mathcal{RE}$ with graph-controlled grammars.

Skipping a Symbol. The rules for this task will have the labels (skip, q, i, j), with $q \in Q$, $1 \leq i \leq \gamma$ and $1 \leq j \leq 11$. More precisely, we take the following rules:

$$\begin{array}{|lllll|}
\hline
((\text{skip},q,i,1): & A \to C, & \{(\text{skip},q,i,2)\}, & \emptyset) \\
((\text{skip},q,i,2): & A \to A, & \{(\text{skip},q,i,3)\}, & \emptyset) \\
((\text{skip},q,i,3): & C \to A, & \{(\text{skip},q,i,4)\}, & \emptyset) \\
((\text{skip},q,i,4): & B \to C^2, & \{(\text{skip},q,i,4)\}, & \{(\text{skip},q,i,5)\}) \\
((\text{skip},q,i,5): & C \to B, & \{(\text{skip},q,i,5)\}, & \{(\text{skip},q,i,6)\}) \\
((\text{skip},q,i,6): & A \to C, & \{(\text{skip},q,i,7)\}, & \{(\text{skip},q,i,11)\}) \\
((\text{skip},q,i,7): & B \to B, & \{(\text{skip},q,i,8)\}, & \emptyset) \\
((\text{skip},q,i,8): & C \to \lambda, & \{(\text{skip},q,i,9)\}, & \emptyset) \\
((\text{skip},q,i,9): & A \to C, & \{(\text{skip},q,i,6)\}, & \{(\text{skip},q,i,10)\}) \\
((\text{skip},q,i,10): & B \to B^2, & \{(\text{skip},q,i,11)\}, & \emptyset) \\
((\text{skip},q,i,11): & C \to A, & \{(\text{skip},q,i,11)\}, & \text{exit-skip}(i)) \\
\hline
\end{array}$$

Here, exit-skip(i) equals $\{(\text{skip}, q, i+1, 1)\}$ if $i < \gamma$ and simstart(q), otherwise.

Simulation. We must give a separate simulation for each of the four possible cases of a Turing machine rule. Fix some rule $r = (q, a, X, q', a') \in \delta$ in the

following. For reasons of space, we only detail on the first case: consider $a \in \tilde{\Gamma}$ (Recall that $\tilde{\Gamma} = \Gamma \setminus \{\#_L, \#_R\}$.) and $X = L$. The other cases can be treated similarly.
Let $\beta(aq) = \beta_1 \dots \beta_{2\gamma}$ with $\beta_i \in \{0,1\}$ and $\beta(q'a') = \beta'_1 \dots \beta'_{2\gamma}$ with $\beta'_i \in \{0,1\}$.

The simulation of a Turing step has two sub-phases: firstly, it is checked whether aq is codified in the current position (which has been reached by repeated applications of the skip procedure), and then $q'a'$ is generated in its place.

We take the following rules in the checking phase:

$$\begin{array}{|l|}\hline
((\text{sim-1}, r, i, 1) : A \to C,\ \{(\text{sim-1}, r, i, 2)\},\ \emptyset) \\
((\text{sim-1}, r, i, 2) : A \to A,\ \{(\text{sim-1}, r, i, 3)\},\ \emptyset) \\
((\text{sim-1}, r, i, 3) : C \to A,\ \{(\text{sim-1}, r, i, 4)\},\ \emptyset) \\
((\text{sim-1}, r, i, 4) : A \to C,\ \{(\text{sim-1}, r, i, 5)\},\ f_{0,i}(\beta_{2\gamma-i+1})) \\
((\text{sim-1}, r, i, 5) : B \to B,\ \{(\text{sim-1}, r, i, 6)\},\ \emptyset) \\
((\text{sim-1}, r, i, 6) : C \to \lambda,\ \{(\text{sim-1}, r, i, 7)\},\ \emptyset) \\
((\text{sim-1}, r, i, 7) : A \to C,\ \{(\text{sim-1}, r, i, 4)\},\ f_{1,i}(\beta_{2\gamma-i+1})) \\
((\text{sim-1}, r, i, 8) : C \to A,\ \{(\text{sim-1}, r, i, 8)\},\ \text{cont-sim-1}(i)) \\ \hline
\end{array}$$

Here, for $b \in \{0,1\}$,

$$f_{j,i}(b) = \begin{cases} \emptyset, & \text{if } j \neq b; \\ \{(\text{sim-1}, r, i, 8)\}, & \text{if } j = b; \end{cases}$$

and

$$\text{cont-sim-1}(i) = \begin{cases} \{(\text{sim-1}, r, i+1, 1)\}, & \text{if } i < 2\gamma; \\ \{(\text{sim-1}, r, 1, 9)\}, & \text{if } i = 2\gamma. \end{cases}$$

Moreover, we take the following rules in the generating phase:

$$\begin{array}{|l|}\hline
((\text{sim-1}, r, i, 9) :\ B \to C^2,\ \{(\text{sim-1}, r, i, 9)\},\ \{(\text{sim-1}, r, i, 10)\}) \\
((\text{sim-1}, r, i, 10) : C \to B,\ \{(\text{sim-1}, r, i, 10)\},\ f'_i(\beta'_i)) \\
((\text{sim-1}, r, i, 11) : B \to B^2,\ \text{exit-sim-1}(i),\ \emptyset) \\ \hline
\end{array}$$

Here,

$$f'_i(b) = \begin{cases} \text{exit-sim-1}(i), & \text{if } b = 0; \\ \{(\text{sim-1}, r, i, 11)\}, & \text{if } b = 1; \end{cases}$$

and exit-sim-1(i) equals $\{(\text{sim-1}, r, i+1, 9)\}$ if $i < 2\gamma$ and $\{(\text{return}, q', 1)\}$, otherwise. In any case, we have $1 \leq i \leq 2\gamma$ and $1 \leq j \leq 11$.

Returning to Standard Presentation. The corresponding rules are simply obtained by interchanging the roles of A and B in the skipping construction. For

$q \in Q$ and $1 \leq j \leq 10$, we take the following rules:

$$\begin{array}{|ll|}\hline
((\text{return}, q, 1): & B \to C, \ \{(\text{return}, q, 2)\}, \emptyset) \\
((\text{return}, q, 2): & B \to B, \ \{(\text{return}, q, 3)\}, \{(\text{return}, q, 10)\}) \\
((\text{return}, q, 3): & C \to B, \ \{(\text{return}, q, 4)\}, \emptyset) \\
((\text{return}, q, 4): & A \to C^2, \{(\text{return}, q, 4)\}, \{(\text{return}, q, 5)\}) \\
((\text{return}, q, 5): & C \to A, \ \{(\text{return}, q, 5)\}, \{(\text{return}, q, 6)\}) \\
((\text{return}, q, 6): & B \to \lambda, \ \{(\text{return}, q, 7)\}, \{(\text{return}, q, 9)\}) \\
((\text{return}, q, 7): & B \to C, \ \{(\text{return}, q, 6)\}, \{(\text{return}, q, 8)\}) \\
((\text{return}, q, 8): & A \to A^2, \{(\text{return}, q, 9)\}, \emptyset) \\
((\text{return}, q, 9): & C \to B, \ \{(\text{return}, q, 9)\}, \{(\text{return}, q, 1)\}) \\
((\text{return}, q, 10): & C \to B, \ \text{simstart}(q), \quad \emptyset) \\ \hline
\end{array}$$

Here,

$$\begin{aligned}
\text{simstart}(q) = {} & \{(\text{skip}, q, 1, 1)\} \\
& \cup \{(\text{sim-1}, r, 1, 1) \mid r \in \delta, \pi_1(r) = q, \pi_3(r) = L\} \\
& \cup \{(\text{sim-2}, r, b, 1, 1) \mid r \in \delta, \pi_1(r) = q, \pi_3(r) = R, b \in \Gamma \setminus \{\#_L\}\} \\
& \cup \{(\text{sim-3}, r, b, 1, 1) \mid r \in \delta, \pi_1(r) = q, \pi_2(r) = \#_L, \pi_3(r) = R, \\
& \quad b \in \Gamma \setminus \{\#_L\}\} \\
& \cup \{(\text{sim-4}, r, 1, 1) \mid r \in \delta, \pi_1(r) = q, \pi_2(r) = \#_R, \pi_3(r) = L\} \\
& \cup \{(\text{term}, \#_R, 0, 0) \mid q = q_f\}.
\end{aligned}$$

(sim-j,...,1,1) is the start rule label for the simulation subprogram of the jth form of a Turing machine rule. Observe that only in the case when the final state has been reached, the first termination rule may be selected as the next rule to be applied after finishing a simulation loop.

Termination Rules. Firstly, we check in some preparatory steps whether there is at least one A and exactly one B in the string. Then, we continue checking for the occurrence of $\#_R$ at the rightmost position of the simulated Turing tape.

$$\begin{array}{|l|}\hline
((\text{term}, \#_R, 0, 0): A \to A, \ \{(\text{term}, \#_R, 0, 1)\}, \emptyset) \\
((\text{term}, \#_R, 0, 1): B \to C, \ \{(\text{term}, \#_R, 0, 2)\}, \emptyset) \\
((\text{term}, \#_R, 0, 2): B \to B, \emptyset, \qquad \{(\text{term}, \#_R, 0, 3)\}) \\
((\text{term}, \#_R, 0, 3): C \to B, \ \{(\text{term}, \#_R, 1, 1)\}, \emptyset) \\ \hline
\end{array}$$

Now, let $\hat{\Sigma} = \Sigma \cup \{\#_L, \#_R, \#, q_f\}$ be the set of symbols admissible in a configuration whose tape contains a terminal string. In addition, for $a \in \hat{\Sigma} \setminus \{\#_L\}$ with $\beta(a) = \beta_1 \ldots \beta_\gamma$, $\beta_i \in \{0, 1\}$, and for $1 \leq i < \gamma$, we have:

$$\begin{array}{|l|}\hline
((\text{term}, a, i, 1): A \to C, \ \{(\text{term}, a, i, 2)\}, \ f_{0,i}(\beta_{\gamma-i+1})) \\
((\text{term}, a, i, 2): B \to B, \{(\text{term}, a, i, 3)\}, \emptyset) \\
((\text{term}, a, i, 3): C \to \lambda, \ \{(\text{term}, a, i, 4)\}, \emptyset) \\
((\text{term}, a, i, 4): A \to C, \ \{(\text{term}, a, i, 1)\}, \ f_{1,i}(\beta_{\gamma-i+1})) \\
((\text{term}, a, i, 5): C \to A, \ \{(\text{term}, a, i, 5)\}, \{(\text{term}, a, i+1, 1)\}) \\ \hline
\end{array}$$

Here, for $b \in \{0,1\}$,

$$f_{j,i}(b) = \begin{cases} \emptyset, & \text{if } j \neq b; \\ \{(\text{term}, a, i, 5)\}, & \text{if } j = b. \end{cases}$$

Similarly, the first bit is finally checked:

$$\boxed{\begin{array}{l} ((\text{term}, a, \gamma, 1) : A \to C,\ \{(\text{term}, a, \gamma, 2)\},\ f_{0,i}(\beta_1)) \\ ((\text{term}, a, \gamma, 2) : B \to B,\ \{(\text{term}, a, \gamma, 3)\},\ \emptyset) \\ ((\text{term}, a, \gamma, 3) : C \to \lambda,\ \{(\text{term}, a, \gamma, 4)\},\ \emptyset) \\ ((\text{term}, a, \gamma, 4) : A \to C,\ \{(\text{term}, a, \gamma, 1)\},\ f_{1,i}(\beta_1)) \end{array}}$$

Then, different things may happen, depending on which tape symbol has been currently read:

$$\boxed{\begin{array}{llll} ((\text{term}, \#_R, \gamma, 5) : & C \to A, & \{(\text{term}, \#_R, \gamma, 5)\}, & \{(\text{term}, q_f, 1, 1)\}) \\ ((\text{term}, q_f, \gamma, 5) : & C \to A, & \{(\text{term}, q_f, \gamma, 5)\}, & T(\{\#_R\})) \\ ((\text{term}, \#, \gamma, 5) : & C \to A, & \{(\text{term}, \#, \gamma, 5)\}, & T(\{\#_R\})) \\ ((\text{term}, \tilde{a}, \gamma, 5) : & C \to A, & \{(\text{term}, \tilde{a}, \gamma, 5)\}, & \{(\text{term}, \tilde{a}, \gamma, 6)\}) \\ ((\text{term}, \tilde{a}, \gamma, 6) : & B \to \tilde{a}B, & T(\{\#, \#_R\}), & \emptyset) \end{array}}$$

where $\tilde{a} \in \Sigma$ and $T(X) = \{(\text{term}, a, 1, 1) \mid a \in \hat{\Sigma} \setminus X\}$ for $X \subset \hat{\Sigma}$.

Finally, we check the codification of the leftmost tape symbol, i.e., $\#_L$, and yield the terminal string if everything was all right up to now.

$$\boxed{\begin{array}{ll} ((\text{term}, \#_L, 1, 1) : A \to C, & \{(\text{term}, \#_L, 1, 2)\}, \emptyset) \\ ((\text{term}, \#_L, 1, 2) : B \to \lambda, & \{(\text{term}, \#_L, 1, 3)\}, \emptyset) \\ ((\text{term}, \#_L, 1, 3) : C \to \lambda, & \{(\text{term}, \#_L, 1, 4)\}, \emptyset) \\ ((\text{term}, \#_L, 1, 4) : A \to A, & \emptyset, \qquad \{(\text{term}, \#_L, 1, 1)\}) \end{array}}$$

4.3 Comments on the Correctness of the Construction

In principle, a configuration c of the simulated Turing machine (which is in state q) is codified by $A^{\beta(c)}B$ at any time before the simulation enters a rule from simstart(q). By induction, it can be shown that

1. a complete loop entering $(\text{skip}, q, 1, 1)$ and heading for some rule from simstart$(q) \setminus \{(\text{skip}, q, 1, 1)\}$ (which does not enter a rule from simstart(q) in-between) converts a string[2] of the form $A^{\beta(wa)}B^{1(\beta(u))^R}$ into a string $A^{\beta(w)}B^{1(\beta(ua))^R}$, where $w, u \in \Gamma^*$, $a \in \Gamma$, unless $a = \#_L$ and $w = \lambda$;
2. the rules whose labels start with sim-i correctly simulate an application of a rule of type i of the Turing machine;
3. a string of the form $A^{\beta(w)}B^{1(\beta(u))^R}$ is correctly converted into a string $A^{\beta(wu)}B$ by repeated applications of rules whose labels start with return;

[2] Here, B^{1x} for some $x \in \{0,1\}^*$ is the string of B's obtained by interpreting the binary string $1x$ as a binary number.

4. (only) a codified tape of the form $c \in \#_L w \#^* q_f \#_R$ for some $w \in \Sigma$, i.e., $A^{\beta(c)}B$ can be correctly transformed into the terminal string w;
5. all sentential forms generatable by the programmed grammar are of the form $A^+(B \cup C)^+ \cup (A \cup C)^* \Sigma^* B^+ \cup \Sigma^*$.

Basically from these considerations together with Remark 1, the correctness of the proposed construction may be inferred.

5 Further Consequences

5.1 Leftmost Derivations

A natural variant of context-free programmed grammars with appearance checking is to consider them working under *leftmost derivations of type 3*, as was already done in the very first paper on programmed grammars [13]. Since the construction in our main theorem can also be viewed in a leftmost fashion, we can conclude:

Corollary 1. $\mathcal{RE} = \mathcal{P}_3^{\ell-3}$.

Meduna and Horváth [11, Theorem 5] previously considered the nonterminal complexity of context-free programmed grammars without appearance checking working under leftmost derivations of type 2 (within Kasai's formalism of state grammars [7]). They claimed that, for these grammars, three nonterminals are enough to generate every language from $\mathcal{RE}$. Unfortunately, the coding trick used in [11, Theorem 5] does not work properly.[3] Slight modifications of our main construction lead to:

Theorem 5. $\mathcal{RE} = \mathcal{P}_4^{\ell-2}$.

Similarly, we can show:

Corollary 2. $\mathcal{RE} = \mathcal{PUT}_4^{\ell-2}$.

5.2 Matrix Grammars

Similarly, one could consider matrix languages instead of programmed languages. Due to [1, Lemma 4.1.4], matrix grammars can simulate programmed grammars at the expense of one additional nonterminal. Therefore, we can state:

Corollary 3. $\mathcal{RE} = \mathcal{M}_4 = \mathcal{M}_4^{\ell-3}$.

This improves the previously published bound of 6 nonterminals for $\mathcal{M}$, see [1]. Freund and Păun obtained a matching result for matrix grammars [5].

[3] The reader who wishes to study the proof of Meduna and Horváth should consider the possibility that a two-letter sentential form AB codified as 01001 in the simulation will yield 00101 after simulating the rule $B \to B$ (if no other applicable rule is in the present "state").

5.3 Unconditional Transfer

We are now going to bound the nonterminal complexity of context-free programmed grammars with unconditional transfer working under leftmost derivations of type 3, which were shown to be computationally complete in [4] by an intrinsically non-constructive argument.

In this section, let $\mathcal{L}(\Sigma)$ denote the restriction of the language class $\mathcal{L}$ on languages over the alphabet Σ.

Recall the notion of division ordering: for $u, v \in \Sigma^*$, $u = u_1 \dots u_n$, $u_i \in \Sigma$, we say that u *divides* v, written $u|v$, if $v \in \Sigma^* u_1 \Sigma^* \dots \Sigma^* u_n \Sigma^*$. u is also called a *sparse subword* of v in this case. The famous Theorem of Higman states that every $L \subseteq \Sigma^*$ has a *finite* subset L' such that every word in L has a sparse subword in L'. If $I(u) = \{v \in \Sigma^* \mid u|v\}$ denotes the ideal of u, then Higman's Theorem gives the following presentation of L: $L = \bigcup_{u \in L'} (L \cap I(u))$. Let us call L' a *Higman basis* of L. This presentation has been one of the ideas for showing the computational completeness of $\mathcal{PUT}^{\ell-3}$. More precisely, it is clear from our quoted construction that the nonterminal complexity of the constructed grammar basically depends on three parameters: (1) the nonterminal complexity of the simulated grammar, (2) the size of the alphabet of the language and (3) the maximal length of a word in a Higman basis of the language. This is still true when thinking about a simulation of $\mathcal{P}^{\ell-3}$ grammars instead of starting from type-0-grammars in Kuroda normal form, as we did in [4]. We will give details of such a construction below. This means that we can consider parameter (1) as a constant due to our main theorem. Since we keep (2) fixed by definition in the following, we only need to worry about (3). The key observation is therefore:

Lemma 1. *Let Σ be an alphabet. Then, there is a constant $n_{|\Sigma|}$ such that every recursively enumerable language $L \subseteq \Sigma^*$ possesses a Higman basis $\hat{L}$ such that every word $w \in \hat{L}$ obeys $|w| \leq n_{|\Sigma|}$.*

A sequence of modifications of the proof of the main theorem yields:

Theorem 6. $\forall \Sigma \exists c > 0 : \mathcal{PUT}_c^{\ell-3}(\Sigma) = \mathcal{RE}(\Sigma)$.

Admittedly, the dependence on the size of the terminal alphabet in the previous theorem appears to be somewhat peculiar and seems to be special to programmed languages with unconditional transfer. Note that the construction used in [4] entails a dependence on the minimal size of a Higman basis of a language. As the example $L_n = \bigcup_{i=1}^{n} \{a_i^j \mid j \geq 1\}$ (with minimal Higman basis $\{a_1, \dots, a_n\}$) shows, the size of minimal Higman bases will grow arbitrarily large with growing terminal alphabet size. It is still an open question whether a bound on the nonterminal complexity of context-free programmed grammars with unconditional transfer working under leftmost derivation of type 3 can be derived without limiting the size of the terminal alphabet.

Similarly, matrix grammars with unconditional transfer can be considered, see [2].

6 Concluding Remarks

We briefly discuss whether our main Theorem 4 and its consequences can be further improved. It is known [1, Theorem 4.2.2] that there exists a regular language in $\mathcal{P}_2 \setminus \mathcal{P}_1$. Hence, the chain $\mathcal{P}_1 \subset \mathcal{P}_2 \subseteq \mathcal{P}_3 = \mathcal{RE}$ poses only one open problem. We conjecture that $\mathcal{P}_2 \subseteq \mathcal{P}_3$ is strict. Similarly, most other bounds derived in this paper are rather sharp. Nonetheless, it would be challenging to really prove these sharpness conjectures.

It would be interesting to know whether a nonterminal complexity bound *dependent* on the size of the nonterminal alphabet can be found for context-free random context grammars with appearance checking. Note that an upper bound independent of the alphabet size does not exist [1, Example 4.1.1].

Acknowledgments. We are grateful for immediate answers of our colleagues H. Bordihn and Gh. Păun concerning questions on syntactic complexity and for some discussions with R. Freund and F. Stephan.

References

1. J. Dassow and Gh. Păun. *Regulated Rewriting in Formal Language Theory.* Berlin: Springer, 1989.
2. H. Fernau. Unconditional transfer in regulated rewriting. *Acta Informatica*, 34:837–857, 1997.
3. H. Fernau. On regulated grammars under leftmost derivation. *GRAMMARS*, 3:37–62, 2000.
4. H. Fernau and F. Stephan. Characterizations of recursively enumerable languages by programmed grammars with unconditional transfer. *Journal of Automata, Languages and Combinatorics*, 4(2):117–142, 1999.
5. R. Freund and Gh. Păun. On the number of non-terminal symbols in graph-controlled, programmed and matrix grammars. This volume.
6. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Reading (MA): Addison-Wesley, 1979.
7. T. Kasai. A hierarchy between context-free and context-sensitive languages. *Journal of Computer and System Sciences*, 4:492–508, 1970.
8. A. Meduna. Syntactic complexity of scattered context grammars. *Acta Informatica*, 32:285–298, 1995.
9. A. Meduna. Four-nonterminal scattered context grammars characterize the family of recursively enumerable languages. *International Journal of Computer Mathematics*, 63:67–83, 1997.
10. A. Meduna. Generative power of three-nonterminal scattered context grammars. *Theoretical Computer Science*, 246:279–284, 2000.
11. A. Meduna and Gy. Horváth. On state grammars. *Acta Cybernetica*, 8:237–245, 1988.
12. Gh. Păun. Six nonterminals are enough for generating each r.e. language by a matrix grammar. *International Journal of Computer Mathematics*, 15:23–37, 1984.
13. D. J. Rosenkrantz. Programmed grammars and classes of formal languages. *Journal of the ACM*, 16(1):107–131, 1969.
14. C. E. Shannon. A universal Turing machine with two internal states. IN: Automata Studies, pages 157–165. Princeton University Press, 1956.

On the Number of Non-terminal Symbols in Graph-Controlled, Programmed and Matrix Grammars

Rudolf Freund[1] and Gheorghe Păun[2,⋆]

[1] Institute for Computer Languages, Vienna University of Technology
Favoritenstraße 9, A-1040 Wien, Austria
rudi@logic.at, rudi@emcc.at
[2] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 70700 Bucureşti, Romania
gpaun@imar.ro, g_paun@hotmail.com, gp@astor.urv.es

Abstract. We improve the results elaborated in [6] on the number of non-terminal symbols needed in matrix grammars, programmed grammars, and graph-controlled grammars with appearance checking for generating arbitrary recursively enumerable languages. Of special interest is the result that the number of non-terminal symbols used in the appearance checking mode can be restricted to two. In the case of graph controlled (and programmed grammars) with appearance checking also the number of non-terminal symbols can be reduced to three (and four, respectively); in the case of matrix grammars with appearance checking we either need four non-terminal symbols with three of them being used in the appearance checking mode or else again we only need two non-terminal symbols being used in the appearance checking mode, but in that case we cannot bound the total number of non-terminal symbols.

1 Introduction

The motivation for the present work comes from two directions. On one hand, it improves an "old" result in the area of regulated rewriting ([6]), of a rather basic nature (minimizing the number of non-terminal symbols necessary in order to generate an arbitrary recursively enumerable language by a matrix grammar with appearance checking). It should be emphasized that the number of non-terminal symbols is a basic descriptional measure of Chomsky grammars and of their extensions. On the other hand, as matrix grammars recently are widely used for proving universality results in the area of *membrane systems* (P systems), see [7], [1], and the number of non-terminal symbols used by matrix grammars in the so-called appearance checking mode is of crucial importance in obtaining universal P-systems with a small number of membranes (see the technique introduced in [4]), this "old" problem becomes of a renewed interest.

⋆ Work supported by Bundesministerium für Bildung, Wissenschaft und Kunst, Austria, grant GZ 55.300/269-VII/D/5a/2000 and by a grant of the NATO Science Committee, Spain, 2000-2001

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 214–225, 2001.

In [6] it was shown that six non-terminal symbols are enough in matrix grammars with appearance checking for generating any arbitrary recursively enumerable language; all six variables were used also in the appearance checking mode. The purpose of this paper is to improve this result on the number of non-terminal symbols, i.e., we shall show that four non-terminal symbols, only three of them being used in the appearance checking mode, are sufficient. If we allow an arbitrary number of non-terminal symbols, then the number of non-terminals being used in the appearance checking mode can even be reduced to two. This result also allows us to establish a stronger version of the binary normal form for matrix grammars with appearance checking which we shall call the strong binary normal form. This special result has already been used in some other papers, especially in the area of P-systems, e.g., see [4]. For graph-controlled and programmed grammars with appearance checking the total number of non-terminal symbols can be reduced to three and four, respectively, and only two of them need to be used in the appearance checking mode anyway.

The basic result we use for proving our main theorems has already been known since many years, i.e., in [5] it was shown that the actions of a Turing machine can be simulated by a register machine with only two registers. The actions of such register machines can easily be simulated by graph-controlled grammars with appearance checking (and programmed and matrix grammars with appearance checking as well); the main difficulty that remains is to generate the terminal string and at the same time also the corresponding encoding for the simulation of the register machine by having only two non-terminal symbols being used in the appearance checking mode.

Independently, similar results for bounding the total number of non-terminal symbols in programmed and matrix grammars with appearance checking have been obtained by Henning Fernau in [3], who uses a tricky Turing machine simulation for obtaining his main result for programmed grammars with appearance checking, where he even needs only three non-terminal symbols, yet all of them are used in the appearance checking mode. On the other hand, he has to use all four non-terminal symbols in the appearance checking mode in his construction for matrix grammars with appearance checking, which result is not optimal for the purposes needed in the area of membrane computing (see [4]).

The rest of the paper is organized as follows: In the next section we give the definitions for arbitrary grammars and graph-controlled, programmed and matrix grammars with appearance checking as well as for register machines. The main result of this paper is established in section three, i.e., we show that for each recursively enumerable language there exists a graph-controlled grammar with appearance checking with only three non-terminal symbols and with only two of them being used in the appearance checking mode. In the fourth section we establish the results for matrix grammars with appearance checking. We conclude with a short summary of the results elaborated in this paper and with some aspects of related and future research.

2 Definitions

In this section we first recall some well-known notions from formal language theory and then define the main control mechanisms to be considered in this paper, i.e., we define graph-controlled, programmed, and matrix grammars with appearance checking. Finally we introduce the model of register machines we are going to simulate for establishing our main results.

2.1 Formal Language Prerequisites

The reader is assumed to be familiar with the basic notions of formal language theory, e.g., see [2] and [8]. We only give the following definitions:

For an alphabet V, by V^* we denote the free monoid generated by V under the operation of concatenation; the *empty word* is denoted by λ, and $V^+ := V^* \setminus \{\lambda\}$. For any word $w \in V^*$ and any $X \in V$, $|w|_X$ represents the number of occurrences of the symbol X in w. The set of non-negative integers is denoted by $\mathbb{N}_0$, and by RE we denote the family of recursively enumerable languages.

2.2 Grammars with Control Mechanisms

In this subsection we give the definitions of the main mechanisms of regulated rewriting we shall consider in this paper, i.e., we define graph-controlled grammars, programmed grammars, and matrix grammars with appearance checking. As in this paper we only deal with such grammars using the appearance checking mode, in the following we shall omit this additional information, thus only speaking of graph-controlled grammars, programmed grammars, and matrix grammars, respectively. As is well known (see [2]), each of these control mechanisms allows us to obtain RE with context-free productions.

Graph-Controlled Grammars. A *graph-controlled grammar* is a construct $G_C = (N, T, (R, L_{in}, L_{fin}), S)$; N and T are sets of *non-terminal* and *terminal symbols,* respectively, with $N \cap T = \emptyset$; $S \in N$ is the start symbol; R is a finite set of rules r of the form $(l(r) : p(l(r)), \sigma(l(r)), \varphi(l(r)))$, where $l(r) \in Lab(G_C)$, $Lab(G_C)$ being a set of labels associated (in a one-to-one manner) with the rules r in R, $p(l(r))$ is a context-free production over $N \cup T$, $\sigma(l(r)) \subseteq Lab(G_C)$ is the *success field* of the rule r, and $\varphi(l(r))$ is the *failure field* of the rule r; $L_{in} \subseteq Lab(G_C)$ is the set of initial labels, and $L_{fin} \subseteq Lab(G_C)$ is the set of final labels. For $r = (l(r) : p(l(r)), \sigma(l(r)), \varphi(l(r)))$ and $v, w \in (N \cup T)^*$ we define $(v, l(r)) \Longrightarrow_{G_C} (w, k)$ if and only if

- **either** $p(l(r))$ is applicable to v, the result of the application of the production $p(l(r))$ to v is w, and $k \in \sigma(l(r))$,
- **or** $p(l(r))$ is not applicable to v, $w = v$, and $k \in \varphi(l(r))$.

The language generated by G_C is

$$L(G_C) = \{w \in T^* \mid (S, l_0) \Longrightarrow_{G_C} (w_1, l_1) \ldots \Longrightarrow_{G_C} (w_k, l_k),\ k \geq 1, \\ w_j \in (N \cup T)^* \text{ and } l_j \in Lab(G_C) \text{ for } 0 \leq j \leq k, \\ w_0 = S,\ w_k = w,\ l_0 \in L_{in},\ l_k \in L_{fin}\}.$$

A non-terminal symbol $A \in N$ is said to be used in the appearance checking mode, if there exists at least one rule $(l : p, \sigma(l), \varphi(l)) \in R$ such that p is of the form $A \to \alpha$ with $\alpha \in (N \cup T)^*$ and $\varphi(l) \neq \emptyset$.

Programmed Grammars. A *programmed grammar* is a construct $G_P = (N, T, R, S)$ such that $G_C = (N, T, (R, Lab(G_C), Lab(G_C)), S)$ is a graph-controlled grammar, where $Lab(G_C)$ is the set of labels associated (in a one-to-one manner) with the rules r in R; i.e., in a programmed grammar we do not specify initial and final rules. As we shall see later on in this paper, the major drawback of programmed grammars in comparison with graph-controlled grammars with respect to our goals is the absence of the possibility to specify initial rules.

Matrix Grammars. A *matrix grammar* is a construct $G_M = (N, T, (M, F), S)$ where N and T are sets of *non-terminal* and *terminal symbols,* respectively, with $N \cap T = \emptyset$, $S \in N$ is the start symbol, M is a finite set of matrices, $M = \{m_i \mid 1 \leq i \leq n\}$, where the matrices m_i are sequences of the form $m_i = (m_{i,1}, \ldots, m_{i,n_i})$, $n_i \geq 1$, $1 \leq i \leq n$, and the $m_{i,j}$, $1 \leq j \leq n_i$, $1 \leq i \leq n$, are context-free productions over $N \cup T$, and F is a subset of $\bigcup_{1 \leq i \leq n,\, 1 \leq j \leq n_i} \{m_{i,j}\}$.

For $m_i = (m_{i,1}, \ldots, m_{i,n_i})$ and $v, w \in (N \cup T)^*$ we define $v \Longrightarrow_{m_i} w$ if and only if there are $w_0, w_1, \ldots, w_{n_i} \in (N \cup T)^*$ such that $w_0 = v$, $w_{n_i} = w$, and for each $j, 1 \leq j \leq n_i$,

- **either** w_j is the result of the application of $m_{i,j}$ to w_{j-1},
- **or** $m_{i,j}$ is not applicable to w_{j-1}, $w_j = w_{j-1}$, and $m_{i,j} \in F$.

The language generated by G_M is

$$L(G_M) = \{w \in T^* \mid S \Longrightarrow_{m_{i_1}} w_1 \ldots \Longrightarrow_{m_{i_k}} w_k,\ w_k = w, \\ w_j \in (N \cup T)^*,\ m_{i_j} \in M \text{ for } 1 \leq j \leq k, k \geq 1\}.$$

A non-terminal symbol $A \in N$ is said to be used in the appearance checking mode, if there exists at least one production of the form $A \to \alpha$, $\alpha \in (N \cup T)^*$, that appears in F.

2.3 Register Machines

In this subsection we introduce the model of register machines that we shall use to establish our main result in the next section.

The main idea of a register machine is to use registers for storing non-negative integers (in a unary representation) and to use very simple operations to change the contents of a register; usually these operations are the addition of one to and the (conditional) subtraction of one from the contents of a register. With these instructions we now can write programs computing functions on natural numbers.

We shall introduce a model which (syntactically) is even more powerful than the models used in [5], hence all the results obtained there with those models can also be obtained with this model:

An *n-register machine* is a construct $M = (n, R, i, f)$, where n is a natural number, i.e., the number of registers the machine may use; R is a set of *labelled program instructions* of the form $k : (op(i), l, m)$ such that $op(i)$ is an operation on register i of M and k, l, m are labels from a set of labels $Lab(M)$ ($Lab(M)$ labels the labelled program instructions of M in a one-to-one manner), $k \neq f$, l is the label for continuing the program if $op(i)$ can be applied to register i and m is the label for continuing the program if it is not possible to apply $op(i)$ to register i; to the *final label* f we assign the program instruction *end*, which halts the program of the register machine M; i is the *initial label* to start the program.

We will use the following program instructions (op, l, m) :

- $(S(i), l, m)$: if possible (i.e., if the contents of register i is greater than zero) subtract one from register i and go to label l, otherwise skip, i.e., do not change the contents of register i, and continue with the instruction at label m;
- $(A(i), h, h)$: add one to register i and continue the program with the instruction at label h; obviously, the operation $A(i)$ is always possible, hence both labels where to continue have to be the same.

In the models used in [5], h as well as one out of $\{l, m\}$ had to be $k + 1$with k being the label of the labelled program instruction under consideration; yet we shall not impose such restrictions on our model here in this paper, because our simulations can deal with these extended features without any problems. Moreover, writing programs for a register machine becomes easier without such restrictions.

An n-register machine M now can be used to compute a partial recursive function $f : \mathbb{N}_0 \to \mathbb{N}_0$ in the following way:

M starts with $m \in \mathbb{N}_0$ in register 1; if M halts in the final label f and with the contents of register 1 being r, then we say that M has computed $f(m) = r$, otherwise, if M does not halt in the final label f when started with m in register 1, then $f(m)$ remains undefined.

As a very simple example to explain our model we construct a 1-register machine computing the partial function $f : \mathbb{N}_0 \to \mathbb{N}_0$ which yields 0 for even numbers and remains undefined for odd natural numbers:

Example 1. Consider the 1-register machine $M = (1, R, 0, 2)$, where

$$R = \{0 : (S(1), 1, 2), 1 : (S(1), 0, 1), 2 : end\}.$$

More intuitively, we can write down M in a program-like style agreeing in the fact that 0 is the initial label:

$0 : (S(1), 1, 2)$
$1 : (S(1), 0, 1)$
$2 : end$

This simple program works as follows: If at label 0 the contents of register 1 is zero, we jump to the end label 2 and M stops; otherwise we subtract one and continue at label 1, where again we try to subtract one; if this is not possible, we enter an infinite loop in 1; otherwise we subtract one and return to label 0. It now is obvious to see that in fact M computes the partial function $f : \mathbb{N}_0 \to \mathbb{N}_0$ which computes 0 for an even number and remains undefined for an odd natural number m.

The main result established in [5] is that the actions of a deterministic Turing machine can be simulated by a 2-register machine:

Let T be an alphabet of (terminal) symbols with $card(T) = z - 1$ and $T = \{a_i \mid 1 \leq i \leq z - 1\}$; then every smbol a_i in T can be interpreted as the digit i at base z; hence, every word in T^* can be encoded as a non-negative integer using the function $g_z : T^* \to \mathbb{N}_0$ defined by $g_z(\lambda) = 0$, $g_z(a_i) = i$ for $1 \leq i \leq z - 1$, and $g_z(wa) = g_z(w) * z + g_z(a)$ for $a \in T$ and $w \in T^*$.

Proposition 1. *(see [5]) Let $L \subseteq T^*$ be a formal language accepted by the Turing machine M_T in such a way that, for every $w \in T^*$, M_T halts if and only if $w \in L$. Then we can construct a 2-register machine M_L such that, for every $w \in T^*$, if M_L starts with $2^{g_z(w)}$ in its first register, the program terminates if and only if M_T halts on w (i.e., if and only if $w \in L$). Moreover, without loss of generality, we may assume that if M_L halts on input $2^{g_z(w)}$, then the contents of both registers is zero.*

3 The Main Result: Simulating Register Machines by Graph-Controlled Grammars

Using the result from [5] that the actions of a (deterministic) Turing machine can be simulated by a register machine with only two registers, i.e., based on Proposition 1, we are able to establish our main result:

Theorem 1. *For each recursively enumerable language L there exists a graph-controlled grammar $G_C = (\{A, B, C\}, T, (R, \{i\}, \{f\}), A)$, with only three non-terminal symbols (A, B, C) and with only two of them (B, C) being used in the appearance checking mode, which generates L.*

Proof. We can only give a sketch of the proof, yet we will specify all necessary ingredients for the graph-controlled grammar G_C.

The main steps of a derivation in G_C can be described as follows:

- Start with the initial rule $(i : A \to A, \sigma(i), \emptyset)$. In general, we may use rules of the form $(k : A \to A, \sigma(k), \emptyset)$ as connections between different modules of the "program" represented by the graph-controlled grammar G_C.
- We now generate a terminal word $w = a_1 ... a_k$ at the beginning of the sentential form in such a way that, when adding a new symbol a_i, in parallel its encoding is generated according to the formula $g_z(wa) = g_z(w) * z + g_z(a)$ using productions $A \to a_i A$ and using the interpretation of the number of non-terminal symbols B and C as the contents of the two registers of a 2-register machine, which finally yields a sentential form $wAB^{g_z(w)}$.
  For example, let $T = \{a\}$, hence, $z = 2$. Then adding one more terminal symbol a is accomplished by the following sequence of rules:
  $(k_1 : A \to A, \{k_1 + 1\}, \emptyset)$
  $(k_1 + 1 : A \to aA, \{k_1 + 2\}, \emptyset)$
  $(k_1 + 2 : B \to \lambda, \{k_1 + 3\}, \{k_1 + 4\})$
  $(k_1 + 3 : A \to ACC, \{k_1 + 2\}, \emptyset)$
  $(k_1 + 4 : C \to \lambda, \{k_1 + 5\}, \{k_1 + 6\})$
  $(k_1 + 5 : A \to AB, \{k_1 + 4\}, \emptyset)$
  $(k_1 + 6 : A \to AB, \{k_1 + 7\}, \emptyset)$
  $(k_1 + 7 : A \to A, \{k_1, k_2\}, \emptyset)$
  In this special case of $z = 2$, in the initial rule we simply may take $\sigma\{i\} = \{k_1\}$, whereas otherwise we have to choose a module for a terminal symbol from T in a non-deterministic way.
- Copy $|v|_B$ to number of symbols A yielding $wA^{g_z(w)+1}$ from $wAB^{g_z(w)}$:
  $(k_2 : A \to A, \{k_2 + 1\}, \emptyset)$
  $(k_2 + 1 : B \to \lambda, \{k_2 + 2\}, \{k_2 + 3\})$
  $(k_2 + 2 : A \to AA, \{k_2 + 1\}, \emptyset)$
  $(k_2 + 3 : A \to A, \{k_3\}, \emptyset)$
- Generate $2^{g_z(w)}$ from $g_z(w)$ in such a way that the sentential form finally is wy with $|y|_B = 2^{g_z(w)-\rho+1}$, $|y|_C = 0$, $|y|_A = \rho$. For a successful derivation, ρ must equal one (this last non-terminal symbol A will be removed at the end of the derivation in G_C); moreover, observe that for $\rho = 0$ we cannot continue successfully, because at least in the last step $k_3 + 8$ of this module we need one non-terminal symbol A:
  $(k_3 : A \to A, \{k_3 + 1\}, \emptyset)$
  $(k_3 + 1 : A \to AB, \{k_3 + 2\}, \emptyset)$
  $(k_3 + 2 : A \to A, \{k_3 + 3, k_3 + 8\}, \emptyset)$
  $(k_3 + 3 : A \to \lambda, \{k_3 + 4\}, \emptyset)$
  $(k_3 + 4 : B \to \lambda, \{k_3 + 5\}, \{k_3 + 6\})$
  $(k_3 + 5 : A \to ACC, \{k_3 + 4\}, \emptyset)$
  $(k_3 + 6 : C \to \lambda, \{k_3 + 7\}, \{k_3 + 2\})$
  $(k_3 + 7 : A \to AB, \{k_3 + 6\}, \emptyset)$
  $(k_3 + 8 : A \to A, \{k_4\}, \emptyset)$

- Simulate 2-register machine M_L (the start label for this simulation of M_L in G_C is k_4, the stop label of M_L has to be identified with label $f-1$ in G_C):
  $k : (S(1), l, m)$ simulated by $(k : B \to \lambda, \{l\}, \{m\})$;
  $k : (S(2), l, m)$ simulated by $(k : C \to \lambda, \{l\}, \{m\})$;
  $k : (A(1), l, l)$ simulated by $(k : A \to AB, \{l\}, \emptyset)$;
  $k : (A(2), l, l)$ simulated by $(k : A \to AC, \{l\}, \emptyset)$.
  Hence, the number of non-terminal symbols B and C represents the contents of register 1 and register 2, respectively, of the 2-register machine M_L.
- If M_L has reached the final state, we finish with
  $(f-1 : A \to \lambda, \{f\}, \emptyset)$ and $(f : B \to \lambda, \emptyset, \emptyset)$.
  The final sentential form is of the form $wA^{\rho-1}$,where ρ is the number of non-terminal symbols A when starting the simulation of the 2-register machine M_L. Hence, for $\rho = 1$ we simply obtain the terminal word $w \in L$ (remember that in this case M_L has halted on $2^{g_z(w)}$). □

Programmed grammars are just a special variant of graph-controlled grammars where we may start and end up with any rule. As already mentioned earlier, the major drawback of programmed grammars in comparison with graph-controlled grammars is the absence of the possibility to specify initial rules. Hence, for the programmed grammars in the succeeding proof we need one more non-terminal symbol just to start the derivation in a correct way:

Corollary 1. *For each recursively enumerable language L there exists a programmed grammar with only four non-terminal symbols and with only two of them being used in the appearance checking mode which generates L.*

Proof. Let $G_C = (\{A, B, C\}, T, (R, \{i\}, \{f\}), A)$ be the graph-controlled grammar constructed in the proof of Theorem 1 with $Lab(G_C)$ being the set of labels associated with the rules in R. Now consider the programmed grammar $G_P = (\{A, B, C, S\}, T, R', S)$ with $Lab(G_P) = Lab(G_C) \cup \{start\}$ being the set of labels associated (in a one-to-one manner) with the rules in R' and *start* being a new label as well as

$$R' = R \cup \{(start : S \to A, \{i\}, \emptyset)\}$$

This new rule $(start : S \to A, \{i\}, \emptyset)$ now allows us to start correctly every derivation in G_P, too. According to the construction given in the proof of Theorem 1 a terminal word w in a derivation in G_C only appears if at the same time we also enter the final state, which guarantees that also in a derivation in G_P a terminal word only appears if w is in L.

Hence, we conclude $L(G_P) = L(G_C) = L$. □

It remains as an open question for future investigations whether the introduction of this special start symbol S could be avoided as claimed by Henning Fernau in [3], yet without having to use all three of them in the appearance checking mode as in the proof of the main theorem in [3].

One of the tricks used in the proof of our main theorem is the final intersection with T^*. On the other hand, the language generated by a graph-controlled grammar could also be defined by only imposing this restriction of distinguishing between non-terminal symbols and terminal symbols by just using the fact that a terminal label is reached as the only selection criterium. In that case, of course, following the proof of Theorem 1, we also have to check that no non-terminal symbol A is left before entering the terminal label. For doing this, we simply may replace the rule $(f : B \to \lambda, \emptyset, \emptyset)$ by $(f : A \to \lambda, \emptyset, \{f+1\})$, which now also uses the non-terminal symbol A in the appearance checking mode, and take $f+1$ as the new terminal label instead of f, where $(f+1 : B \to \lambda, \emptyset, \emptyset)$. In that way we obtain the following result:

Corollary 2. *For each recursively enumerable language L there exists a graph-controlled grammar $G_C = (\{A, B, C\}, T, (R, \{i\}, \{f+1\}), A)$ with only three non-terminal symbols, all of them being used in the appearance checking mode, which generates L in such a way that the final label $f+1$ is reached if and only if the current sentential form is a terminal word (and therefore in L).*

4 Normal Forms for Matrix Grammars

In this section we establish the results for matrix grammars. We first recall some interesting results from [2]:

A matrix grammar $G = (N, T, (M, F), S)$ is said to be in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \#\}$, with these three sets being 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. $(X \to Y, A \to \alpha)$, with $X, Y \in N_1$, $A \in N_2$, $\alpha \in (N_2 \cup T)^*$,
3. $(X \to Y, A \to \#)$, with $X, Y \in N_1$, $A \in N_2$,
4. $(X \to \lambda, A \to \alpha)$, with $X \in N_1$, $A \in N_2$, and $\alpha \in T^*$.

Moreover, there is only one matrix of type 1 and F consists exactly of all rules $A \to \#$ appearing in matrices of type 3; $\#$ is a trap-symbol; once introduced, it is never removed. A matrix of type 4 is used only once, in the last step of a derivation.

According to Lemma 1.3.7 in [2], for each matrix grammar G there is an equivalent matrix grammar G' in the binary normal form.

For an arbitrary matrix grammar $G = (N, T, (M, F), S)$, let us denote by $ac(G)$ the cardinality of the set $\{A \in N \mid A \to \alpha \in F\}$. From the construction in the proof of Lemma 1.3.7 in [2] one can see that if we start from a matrix grammar G and we get the grammar G' in the binary normal form, then $ac(G') = ac(G)$.

A matrix grammar $G = (N, T, (M, F), S)$ is said to be in *strong binary normal form,* if $N = N_1 \cup N_2 \cup \{S, \#\}$, with these three sets being 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. $(X \to Y, A \to \alpha)$, with $X, Y \in N_1$, $A \in N_2$, $\alpha \in (N_2 \cup T)^*$,
3. $(X \to Y, A \to \#)$, with $X, Y \in N_1$, $A \in N_2$,
4. $(X \to \lambda)$, with $X \in N_1$.

Moreover, there is only one matrix of type 1 and only one of type 4 and F consists exactly of all rules $A \to \#$ appearing in matrices of type 3, where $\#$ is the trap-symbol (i.e., once introduced, it can never be removed again). The matrix of type 4 is used only once, in the last step of a derivation. Finally, as the most important feature, we have $ac(G) \leq 2$.

The strong normal form theorem for matrix grammars now again directly follows from Theorem 1:

Theorem 2. *For each recursively enumerable language L there exists a matrix grammar in the strong binary normal form that generates L.*

Proof. Let $G_C = (\{A, B, C\}, T, (R, \{i\}, \{f\}), A)$ be the graph-controlled grammar constructed in the proof of Theorem 1 with $Lab(G_C)$ being the set of labels associated with the rules in R. Now consider the matrix grammar $G_M = (N, T, (M, F), S)$ with $N = N_1 \cup N_2 \cup \{S, \#\}$, $N_1 = Lab(G_C)$, $N_2 = \{A, B, C\}$, $F = \{B \to \#, C \to \#\}$, and M consisting of the following matrices:

- $(S \to iA)$;
- $(X \to Y, D \to \alpha)$, for $(X : D \to \alpha, \sigma(X), \varphi(X)) \in R$ and $Y \in \sigma(X)$ with $X, Y \in N_1$, $D \in N_2$, $\alpha \in (N_2 \cup T)^*$;
- $(X \to Y, D \to \#)$, for $(X : D \to \alpha, \sigma(X), \varphi(X)) \in R$ and $Y \in \varphi(X)$ with $X, Y \in N_1$, $D \in N_2$, $\alpha \in (N_2 \cup T)^*$;
- $(f \to \lambda)$.

By construction, G_M is in the strong binary normal form, and obviously simulates the graph-controlled grammar constructed for L in the proof of Theorem 1; these observations complete the proof. □

We should like to mention that the final matrix in a matrix grammar in the strong binary normal form could also be of the form $(X \to \lambda, A \to \alpha)$ with $X \in N_1$, $A \in N_2$, and $\alpha \in T^*$ like for the binary normal form as defined in [2]. This can easily be achieved due to the construction of the graph-controlled grammar in the proof of Theorem 1, because in a successful derivation leading to a terminal word a non-terminal A is deleted in the last step leading to the terminal label. Hence, if f' is the label of this rule $(f' : A \to \lambda, \{f\}, \emptyset)$, then we simply take $(f' \to \lambda, A \to \lambda)$ as the final matrix in this variant of a strong binary normal form for matrix grammars.

Finally, we can restrict the length of the word α in a matrix $(X \to Y, D \to \alpha)$, with $X, Y \in N_1$, $D \in N_2$, and $\alpha \in (N_2 \cup T)^*$ in a matrix grammar in the strong binary normal form to two as elaborated in [2] for the binary normal form; we leave a proof for this variant to the interested reader.

If we also want to bound the total number of non-terminal symbols in a matrix grammar, we have to pay the price for that by using one more non-terminal symbol in the appearance checking mode: We may use a special non-terminal symbol D to encode the elements of the label alphabet N_1 by suitable powers of D, thus taking advantage of the technique already used in [6] to control the sequence of derivation steps in the matrix grammar. Moreover, if $g-1$ is the maximal power for encoding the labels in N_1, then during a successful derivation leading to a terminal word, at no time more than $g-1$ symbols can be present in a sentential form of such a derivation; therefore D^g can be used as a kind of "trap symbol" as exhibited in [6]. Hence, we immediately obtain the following result:

Theorem 3. *For each recursively enumerable language L there exists a matrix grammar $G_M = (\{A,B,C,D\}, T, (M,F), B)$ with $card(F) \leq 3$ that generates L.*

Proof. Let $G_C = (\{A,B,C\}, T, (R, \{i\}, \{f\}), A)$ be the graph-controlled grammar constructed in the proof of Theorem 1, where without loss of generality we may assume $Lab(G_C) = \{j \mid 1 \leq j \leq g-1\}$ as well as $i = 1$ and $f = g-1$. Now denote a sequence of m equal productions p in a matrix by $(p)^m$. Then, from G_C we construct the matrix grammar $G_M = (N' \cup \{D\}, T, (M,F), B)$ with $N' = \{A,B,C\}$, $F = \{X \to D^g \mid X \in \{B,C,D\}\}$, and M containing the following matrices:

- $(D \to D^g, B \to DA)$ is the start matrix;
- $\left((D \to \lambda)^i, D \to D^g, A \to AD^j, X \to \alpha\right)$ for $(i : X \to \alpha, \sigma(i), \varphi(i)) \in R$ with $X \in N'$, $\alpha \in (N' \cup T)^*$, and $j \in \sigma(i)$;
- $\left((D \to \lambda)^i, D \to D^g, A \to AD^j, X \to D^g\right)$ for $(i : X \to \alpha, \sigma(i), \varphi(i)) \in R$ with $X \in \{B,C\}$, $\alpha \in (N' \cup T)^*$, and $j \in \varphi(i)$;
- $\left((D \to \lambda)^{g-1}, D \to D^g\right)$ is the final matrix.

The start matrix has to be applied as the first matrix in a successful derivation; a rule $D \to D^g$ must not be applied in a derivation that should lead to a terminal word in L, because in every matrix at most $g-1$ symbols D can be removed, before the generation of g symbols D would be enforced again. As at least one copy of the non-terminal symbol A is present in each sentential form of a derivation of a word $w \in L$ in the graph-controlled grammar G_C except for the first and the last one, it is guaranteed that one of these occurrences of A can be used for the productions $A \to AD^j$, which allows us to simulate G_C by G_M in a correct way.

Finally, after the application of the final matrix removing the control symbol D, the initial matrix cannot be applied again, because after the application of the terminal matrix no non-terminal symbol B can be present any more in the underlying sentential form. This observation completes the proof. □

5 Conclusion

We have shown that all recursively enumerable languages can be generated by matrix grammars, programmed grammars, and graph-controlled grammars, respectively, with appearance checking with only two non-terminal symbols being used in the appearance checking mode. If we also want to bound the total number of non-terminal symbols, then we have to pay off by needing one more non-terminal symbol to be used in the appearing checking mode in the case of matrix grammars. The total number of non-terminal symbols can even be restricted to three in graph-controlled grammars, but only to four in the case of programmed grammars and matrix grammars; whether this bound can be reduced to three again for programmed grammars as shown by Henning Fernau in [3] without having to use all three of them in the appearance checking mode, remains for future research. Moreover, in a similar way as done by Henning Fernau in [3], all our results can be carried over from graph-controlled, programmed, and matrix grammars with appearance checking working under free derivations to the corresponding types of grammars working under leftmost derivations.

Acknowledgement

We gratefully acknowledge some interesting discussions with Henning Fernau.

References

1. C. Calude and Gh. Păun. *Computing with Cells and Atoms.* Taylor and Francis, London, 2000.
2. J. Dassow and Gh. Păun. *Regulated Rewriting in Formal Language Theory.* Springer-Verlag, Berlin, 1989.
3. H. Fernau. Nonterminal complexity of programmed grammars. *This volume.*
4. R. Freund and Gh. Păun. Computing with Membranes: Three More Collapsing Hierarchies. *Submitted,* 2000 (also presented at the EMCC Meeting in Milano, Italy, November 3^{rd}, 2000).
5. M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, New Jersey, USA.
6. Gh. Păun. Six Nonterminals are Enough for Generating Each R.E. Language by a Matrix Grammar. *Intern. J. Computer Math.* **15** (1984), pp. 23–37.
7. Gh. Păun. Computing with Membranes. *Journal of Computer and System Sciences* **1**, 1 (2000), pp. 108–143.
8. G. Rozenberg and A. Salomaa (eds.). *Handbook of Formal Languages,* 3 volumes. Springer-Verlag, Berlin, 1997.

A Direct Construction of a Universal Extended H System

Pierluigi Frisco

L.I.A.C.S., Leiden University,
Niels Bohwerg 1, 2333 CA Leiden, The Netherlands
pier@liacs.nl

Abstract. A direct universal extended H system receives as input the coding of an extended H system with double splicing and simulates it. It is the first time that a direct construction is described: universal results obtained until now were based on the simulation of universal type-0 grammars or Turing machines.

1 Introduction

DNA based computation, generally speaking, considers the transformation of biological molecules as computational steps. Recently, many different computability models have been proposed under the inspiration of such biological processes. Splicing systems (see [9], [12]) are a generative mechanism based on the splicing operation as a model of DNA recombination. If these systems have finite sets of axioms and rules defining splicing they can generate regular languages (see details in [15]). Keeping both sets finite the generative power can only be increased by introducing control systems. In this way splicing systems can generate recursive enumerable languages.

A computability model $\mathcal{C}$ is computationally complete if the devices in $\mathcal{C}$ have the power of Turing machines (or of any other type of equivalent device), that is they generate and/or recognize recursive enumerable languages.

There are several systems based on splicing that are computationally compete. For all control systems the demonstration is based on the simulation of type-0 grammars or Turing machines (see [6] and [2] for permitting and forbidding context; [13] for target languages, [14] for programmed and evolving H systems, [11] for double splicing, [3] and [6] for multisets).

Another property that $\mathcal{C}$ may have is universality. This is related to the existence of a fixed element in $\mathcal{C}$ which is able to simulate any other given device in $\mathcal{C}$. Such an element is called *universal*. The first description of a universal Turing machine was given by Turing himself in [10]. Simplified models of universal Turing machine can be found in [17] and [16].

Any model $\mathcal{C}$ that is computationally complete has a universal device. In fact, this is the device that is able to simulate the universal Turing machine, which, in turn, may simulate any device from $\mathcal{C}$.

It is clear that, in the above sentence, splicing systems are universal (explicit constructions were made in [1], [8], [5] and [4]). However this fact is obtained

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 226–239, 2001.

with an *indirect construction*: the universal splicing system receives as input a coding of a Turing machine (for instance) and if this machine is universal, also the splicing system will be universal. In this way, if the input of the simulated universal Turing machine is the coding of a splicing system, we will have a system based on splicing simulating another one.

If a splicing system receives as input the coding of another extended H system and simulates it we will have a *direct construction*.

So the difference between indirect and direct construction relies on the nature of the input: if it is the coding of a splicing system we will have a direct construction, if it is the coding of another kind of computability models we will have indirect construction.

The description of a 'small' direct universal extended H system was indicated as research topic in [15]. In this paper we describe a direct construction of a universal splicing system and we will see how its size is smaller than the ones of the indirect universal extended H system.

2 An Overview of Splicing

We give definitions strictly related to our work; more general information may be found in [15]. Consider an alphabet V and two special symbols, # and \$ not in V. A *splicing rule* is a string of the form $r = u_1\#u_2\$u_3\#u_4$, where $u_1, u_2, u_3, u_4 \in V^*$; u_1u_2 and u_3u_4 are called *sites* of the splicing rule. For such a splicing rule r and strings $x, y, z, w \in V^*$ we write:

$$\begin{aligned}(x, y) \vdash_r (z, w) \text{ iff } & x = x_1u_1u_2x_2, y = y_1u_3u_4y_2,\\ & z = x_1u_1u_4y_2, w = y_1u_3u_2x_2,\\ & \text{for some } x_1, x_2, y_1, y_2 \in V^*,\end{aligned}$$

indicating that x and y splice according to r giving z and w.

Based on this operation, the notion of an *H scheme* is 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

$$\begin{aligned}\sigma(L) &= \{z \in V^* \mid (x, y) \vdash_r (z, w) \text{ or } (x, y) \vdash_r (w, z),\\ &\qquad \text{for some } x, y \in L, r \in R, w \in V^*\},\\ \sigma^0(L) &= L,\\ \sigma^{i+1}(L) &= \sigma^i(L) \cup \sigma(\sigma^i(L)),\ i \geq 0,\\ \sigma^*(L) &= \bigcup_{i\geq 0} \sigma^i(L).\end{aligned}$$

If we consider two families of languages FL_1 and FL_2, we define the family of splicing languages with FL_1 axioms and FL_2 rules, as:

$$H(FL_1, FL_2) = \{\sigma^*(L) \mid L \in FL_1 \text{ and } \sigma = (V, R), R \in FL_2\}.$$

We denote by FIN, REG the families of finite and of regular languages respectively. We have (see details in [15])

$$FIN \subset H(FIN, FIN) \subset REG.$$

An *extended H system* is a construct $\gamma = (V, T, A, R)$, where $\sigma = (V, R)$ is an H scheme and T is an alphabets so that $T \subseteq V$ (T is called *terminal* alphabet), A is a language over V (A is the set of *axioms*). The language generated by γ is $L(\gamma) = \sigma^*(A) \cap T^*$.

If both sets A and R are finite, then $L(\gamma)$ is regular.

If we want an H system having A and R finite generating more than regular languages, then a control system has to be added to the splicing operation. There are many classes of controlled systems and most of them characterize the family of recursively enumerable languages (see [6] and [15]).

The type of control we choose in order to implement our construction is called *double splicing*. An extended H system with double splicing (EHSds) is defined as extended H system where the two strings obtained from a splicing are immediately used in another one.

Formally, if we consider an extended H system $\gamma = (V, T, A, R)$ where x, y, z, $w \in V^*$ and $r_1, r_2 \in R$ we write:

$$(x, y) \vdash_{r_1, r_2} (w, z) \text{ iff } \exists u, v \in V^* \mid (x, y) \vdash_{r_1} (u, v) \text{ and } (u, v) \vdash_{r_2} (w, z)$$

For a language $L \subseteq V^*$ we define

$$\sigma_d(L) = \{z \in V^* \mid (x, y) \vdash_{r_1, r_2} (z, w) \text{ or } (x, y) \vdash_{r_1, r_2} (w, z), \\ \text{for } x, y \in L, r_1, r_2 \in R, w \in V^*\},$$

The definition of σ_d^* is similar to the one of σ^* given above. We associate to γ the language $L_d(\gamma) = \sigma_d^*(A) \cap T^*$.

Given two families of languages FL_1 and FL_2 we denote with $EH_d(FL_1,FL_2)$ the family of languages generated as above by extended H systems with double splicing $\gamma = (V, T, A, R)$ so that $A \in FL_1$ and $R \in FL_2$. $EH_d([k], FL_2)$ indicates the family of languages generated by extended H systems with double splicing having $A \in FIN, card(A) = k$ and $R \in FL_2$ ($card(A)$ is the number of elements of A).

3 Overview of the Construction

A direct universal extended H system has its own splicing rules and axioms and additionally receives as input other axioms defining the system to simulate. These input axioms define both the splicing rules and the axioms of the simulated system. Applying its own splicing rules to its axioms and to the ones received as input the universal system simulates the input system. The language obtained by the universal system can be a code of the one of the simulated one.

When we began to think to which system to use to implement the idea of simulation of splicing we had, we analyzed all kinds of extended H system. Obviously all of them share the operation of splicing but some add to this operation another one that is the control system itself. For instance in an extended H system with permitted context (see [15] for details) the presence of the permitted string(s) related to each rule has to be checked.

The advantage of the double splicing is that the control is obtained without introducing auxiliary operations - one merely has to keep track of the two steps cycle during computation.

For this simplicity we have chosen double splicing as our control mechanism. Moreover the basic operation, rotation of strings, is easy to implement with it.

First splicing	**Second splicing**
1.0 Creation of pairs of axioms	
1.1 Rotation of pairs of axioms	**2.1** Rotation of pairs of axioms
1.2 Rotation of substrings	**2.2** Rotation of substrings
1.3 Join rule-pair of axioms (a working string is obtained)	**2.3** Join rule-pair of axioms (a working string is obtained)
1.4 Rotation of a working string	**2.4** Rotation of a working string
1.5 Matching of the first splicing site	**2.5** Matching of the first splicing site
1.6 Marking of a splicing site	**2.6** Marking of a splicing site
1.7 Wrong match rule-pair of axioms	**2.7** Wrong match rule-pair of axioms
1.8 End of matching. Ordering	**2.8** End of matching. Ordering
1.9 Matching of the second splicing site	**2.9** Matching of the second splicing site
1.10 Marking of a splicing site	**2.10** Marking of a splicing site
1.11 Wrong match rule-pair of axioms	**2.11** Wrong match rule-pair of axioms
1.12 End of matching. Ordering	**2.12** End of matching. Ordering
1.13 Splicing	**2.13** Splicing
	2.14 End simulation

Fig. 1. Structure of the universal system

Most part of the work of the universal system will be done at the ends of strings. These ends have normally the form h_β, t_β (*head* and *tail*) where β defines the *state* of the string. Symbols s (with subscripts) present in strings, are used to separate different logical parts of a string.

Moreover to *rotate* a string will mean to move symbols between h and t (with whatever subscript) from left to right or from right to left. In this context the symbols s are important to keep track of the begin and the end of the rotated substring. For instance the clockwise rotation of two symbols of the string $hs_bv_1v_2v_3s_et$ will bring to $hv_3s_es_bv_1v_2t$. Note how, even when rotated, the "substring" $v_1v_2v_3$ from s_b to s_e is unchanged.

Basic operation. Rotation of a string (together with substitution, deletion or insertion of one or two rotated symbols) is the basic step in the system we describe below. It is performed using a single double splicing. Here we give an

example for anti-clockwise rotation with substitution of a symbol. The generalization to the other cases is easy and left to the reader.

Let us consider the string $h_1abs_1ct_1$ which we want to rotate anti-clockwise by one symbol, changing a into A. In this way we obtain $h_1bs_1cAt_1$. To do this we use the string h_1At_1 present as an axiom in the system.

The operation is performed by one double splicing presented below where both splicing sites are indicated by vertical bars (|):

$$\begin{array}{ccccc} h_1a \mid bs_1ct_1 & & h_1bs_1c \mid t_1 & & h_1bs_1cAt_1 \\ h_1 \mid At_1 & \vdash_1 & h_1a \mid At_1 & \vdash_2 & h_1at_1 \end{array}$$

Firstly the rule 1 : $(h_1a\#\$h_1\#At_1)$ is used and h_1a is replaced with h_1. Now the second splicing operation has to be performed between the two strings obtained by the previous splicing. The rule 2 = $(h_1a\#At_1\$\#t_1)$ changes t_1 into At_1 obtaining the string we wanted.

The universal system simulates a double splicing of the system given as input by repeating several similar operations, see Figure 1. Strings will have different states according to the basic steps they are involved in. Now we to give an overview of the simulation algorithm indicating the different states of strings during all the process.

Single axioms. Axioms of the simulated system will be coded in the universal system by strings of the type:

$$h_1 \boxed{b\ldots\ \ldots} s_1t_1$$

Dots present in the box indicate the code of the axiom of the simulated systems. The other symbols are used by the universal system for internal purposes. The symbol b indicates the *begin* of the code of an axiom (its use will be clear later when this kind of strings are rotated); s's (with any subscript) are used as separators of logical parts of a string when they are joined.

Rules. Also splicing rules of the simulated system are translated into strings in the universal one:

$$\overset{site_1}{} \qquad \overset{site_2}{}$$
$$h_9 \boxed{\ldots} \bar{\#} \boxed{\ldots} \bar{\$} \boxed{\ldots} \bar{\#} \boxed{\ldots} t_9$$

$site_1$ and $site_2$ indicate the two splicing sites in a rule. Each symbol α is represented in the rule by its barred copy $\bar{\alpha}$ to distinguish it from the symbols in the axioms.

1.0 Creation of pairs of axioms. The simulation of a double splicing begins by joining in a non deterministic way two strings representing axioms of the simulated system. The first splicing is performed on this pair of strings and the second on the result of the first splicing.

$$h_1 \boxed{b\ldots\ \ldots} s_1 t_1$$

$$\Rightarrow \qquad \overset{axiom_1}{h_3 \boxed{b\ldots\ \ldots}} s_1 \overset{axiom_2}{\boxed{b\ldots\ \ldots}} s_1 t_3$$

$$h_1 \boxed{b\ldots\ \ldots} s_1 t_1$$

The indications $axiom_1$ and $axiom_2$ will be used from now on to keep track of the two axioms present in a pair.

1.1 Rotation of pairs of axioms. As said before, rotation is the basic operation in the universal system. The next picture illustrates how the rightmost s_1 is moved to the left and then, with the iteration of the process, how the order of the two axioms can be interchanged. In this way such strings can have any of their circular permutation and this is important when a rule is joined to them (paragraph 1.3).

$$\begin{array}{c} \overset{axiom_1}{h_3 \boxed{b\ldots\ \ldots}} s_1 \overset{axiom_2}{\boxed{b\ldots\ \ldots}} s_1 t_3 \\ \Downarrow \\ h_3 s_1 \overset{axiom_1}{\boxed{b\ldots\ldots}} s_1 \overset{axiom_2}{\boxed{b\ldots\ \ldots}} t_3 \\ \Downarrow \\ \ldots \\ \Downarrow \\ h_3 \overset{axiom_2}{\boxed{b\ldots\ \ldots}} s_1 \overset{axiom_1}{\boxed{b\ldots\ \ldots}} s_1 t_3 \end{array}$$

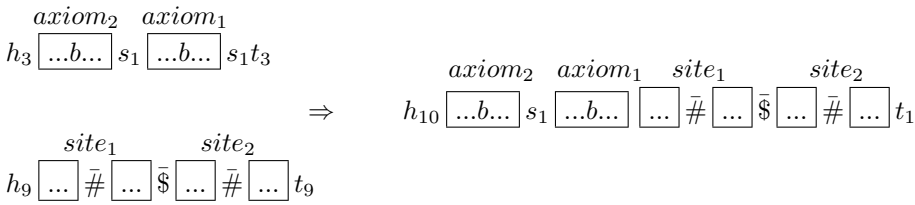
1.2 Rotation of substrings. A substring can be rotated with respect to the logical part of the string it occupies. In the case of a pair of axioms, each of the axioms is such a logical part and forms a substring delimited by s_1's.

The next picture illustrates the clockwise rotation of a symbol β in $axiom_2$. Now it is possible to understand the presence of b and s_1 in a pair of axioms. If both symbols were not present it would be impossible to read the two axioms in the proper way after rotation.

$$\overset{axiom_1}{h_3 \boxed{b\ldots\ \ldots}} s_1 \overset{axiom_2}{\boxed{b\ldots\ \ldots\alpha\beta}} s_1 t_3 \Rightarrow \overset{axiom_1}{h_3 \boxed{b\ldots\ \ldots}} s_1 \overset{axiom_2}{\boxed{\beta b\ldots\ \ldots\alpha}} s_1 t_3$$

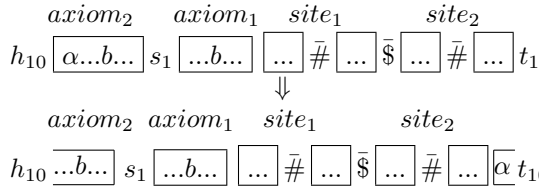
The rotation of a substring is made in a non deterministic way. Thanks to this process the representation of a pair of axioms can have its two axioms in any possible circular permutation, this is important when a rule is joined to them (next paragraph).

1.3 Join a rule to a pair of axioms (a working string is obtained). In a non deterministic way a string representing a rule can be joined with a pair of axioms as to form a *working string*. It is on this kind of strings that the simulation of a double splicing is performed.

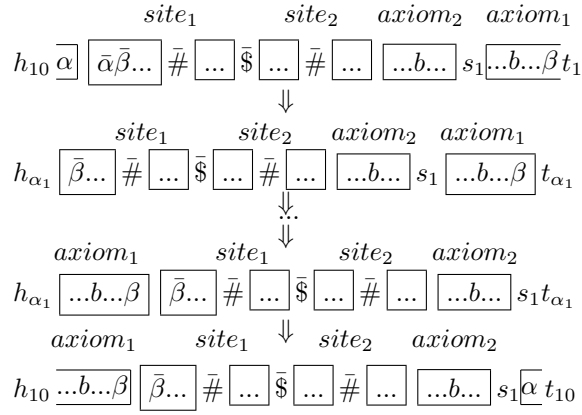


Note that only one s_1 is present in the working string, the other one has been removed during the join operation. Axioms and splicing sites are adjacent: no separator is present between them.

1.4 Rotation of a working string. As for a pair of axioms, a working string can be rotated. This is important to match splicing sites with axioms: particular groups of symbols present at the two ends of a working string are used in order to simulate the splicing.



1.5 Matching of the first splicing site. The symbols of the rule are matched, one by one, with the symbols in the axiom at the splicing site. Rotating a working string it is possible to move $\alpha\bar{\alpha}$ close to the head of the string (we describe only this case, it also possible to have $\bar{\alpha}\alpha$ close to the tail). In this case (showed in the next picture) the matching axiom is rotated by one symbol (α) and the barred symbol in the splicing site is removed. When this operation is completed the working string is ready to match the next symbols (in the example $\bar{\beta}$ and β respectively).

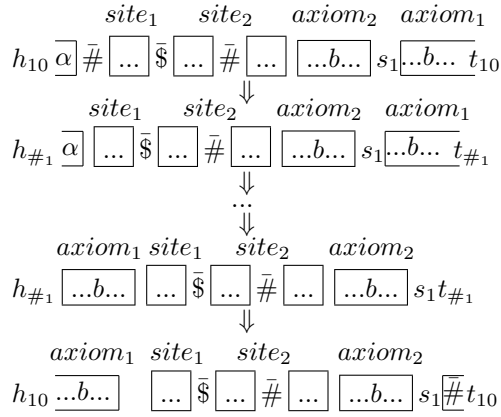


This operation is performed through different steps. When a couple $\alpha\bar{\alpha}$ is present both symbols are removed and the state of the working string in changed

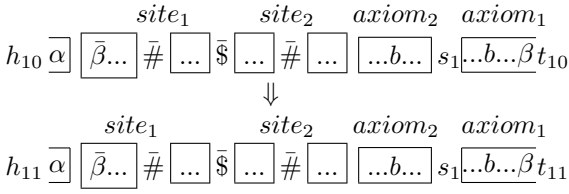
into $\alpha_i, i = 1, 2, 3, 4$ (the subscript of α is used to refer to the matching that is going on: 1 or 2 for the the first or second match of the first splicing, 3 or 4 for the first and second match of the second splicing).

Then the working string is rotated clockwise until s_1 is present at one end. At this point α is added on the right of s_1 and the state of the string in changed back into 10. The process of matching continues rotating the working string.

1.6 Marking a splicing site. When the first part of the rule is matched, $\alpha\bar{\#}$ is met on the left side of the working string (we can also have $\bar{\#}\alpha$ on the right side and the process is similar to the one that we describe). At this point $\bar{\#}$ remains in the checked axiom to keep track of the position where the splicing has to occur, the operation of matching goes on with the rest of the splicing site (if present). Anyhow the $\bar{\#}$ has to be moved in respect to the checked axiom so to permit the continuation of the matching. This is done removing $\bar{\#}$ from the splicing site ($site_1$ in our example) and changing the state into $\#_i, i = 1, 2, 3, 4$ (1 or 2 for the first or second match of the first splicing, 3 or 4 for the first or second match of the second splicing). Then the string is rotated clockwise until s_1 is close to the tail. At this point $\bar{\#}$ is added on the right of s_1 and the state of the string is changed back into 10. The process of matching can go on.



1.7 Wrong match rule-pair of axioms. Of course it is also possible to have a wrong match between a rule and a pair of axioms. This is detected by the presence of $\alpha\bar{\beta}$ where $\alpha \neq \beta$. In this case the working string changes state into 11 and it cannot take part in any further splicing.



1.8 End of matching. Ordering. When the match between $axiom_1$ and $site_1$ is completed the working string will have $\bar{\$}$ on its extreme left (as all symbols present in the splicing site have been consumed). The operation to perform now is ordering, it means to rotate the just matched axiom so to have the b on its extreme right (extreme left for the second match). This is necessary for the implementation of the splicing operation. The ordering is made in a way similar to the rotation of substrings (paragraph 1.2).

$$
\begin{array}{c}
\quad\quad site_2 \quad\quad axiom_2 \quad\quad axiom_1 \\
h_{10}\bar{\$}\boxed{...}\bar{\#}\boxed{...}\ \boxed{...b...}\ s_1\ \boxed{...b...\bar{\#}...}\ t_{10} \\
\Downarrow \\
... \\
\Downarrow \\
\quad\quad site_2 \quad\quad axiom_2 \quad\quad axiom_1 \\
h_{10}\bar{\$}\boxed{...}\bar{\#}\boxed{...}\ \boxed{...b...}\ s_1\ \boxed{...\bar{\#}...b}\ t_{10}
\end{array}
$$

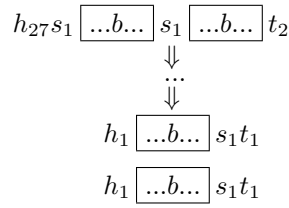
The matching of the second splicing site for the first splicing is symmetric to the first matching just described. Encountered situations are similar but mirrored: what present on the left for the first match is present on the right for the second and vice versa, if the rotation is clockwise for the first match it will be anti-clockwise for the second and vice versa.

1.13 Splicing. If the second splicing site matches the other axiom, after ordering, the working string is ready to simulate the first splicing. The situation will be similar to the one indicated in the next picture. The splicing is simulated in a really simple way: substituting and removing symbols. b's are removed, s_1's are changed into b's and $\bar{\#}$'s into s_1's. The obtained string indicates the result of the simulation of the first splicing written with the syntax used until now: s_1's separate the two substrings and b's indicate their begin.

This obtained string is similar to the pair of axioms created in the paragraph 1.0, so the operations performed for the second splicing are similar to the ones for the first one. Only the states of the working string will be different.

$$
\begin{array}{c}
\quad\quad axiom_2 \quad\quad axiom_1 \\
h_{17}\ \boxed{b...\bar{\#}...}\ s_1\ \boxed{...\bar{\#}...b}\ s_1 t_{17} \\
\Downarrow \\
... \\
\Downarrow \\
h_{19}\ \overline{...}\ s_1\ \boxed{...b...}\ s_1\ \overline{...b}\ t_{19}
\end{array}
$$

2.14 End simulation. After the second splicing the two substrings are separated in two independent strings in state 1. A double splicing has been performed and the two resulting strings can be involved in a new one.



4 Inside the Algorithm

In order to define a universal EHSds we need to consider the coding of an arbitrary EHSds as input for the universal system. As the universal EHSds has a fixed, finite, alphabet, our coding scheme must allow the encoding of arbitrary alphabets. Additionally, the coding transforms both axioms and rules of a system into additional axioms for the universal system. The coding must be "fair" i.e., it should be simple and not solve computational problems beforehand.

Let $\gamma_u = (V_u, T_u, A_u, R_u)$ and $\gamma = (V, T, A, R)$ be EHSds's. Moreover consider the functions:

$$C_A : V^* \to V_u^*,$$
$$C_R : V^*\#V^*\$V^*\#V^* \to V_u^*$$
$$D : T_u^* \to T^*$$

where C_A transforms axiom in A into axioms in A_u; C_R transforms rules of R into axioms in A_u and D transforms final strings of the universal system into final strings of the simulated one.

The system γ_u is called *universal* if it generates the language of γ, when we add the axioms and rules of γ to γ_u (under a suitable coding).

Formally, γ_u is universal iff, for each EHSds γ, $D(L(\gamma_u(\gamma))) = L(\gamma)$ where $\gamma_u(\gamma) = (V_u, T_u, A_u \cup C_A(A) \cup C_R(R), R_u)$.

Functions C_A, C_R and D. We say that a string $q \in V_u^*$ is in *state* l_1 if $q = h_{l_1}...h_{l_d}et_{l_d}...t_{l_1}$ while no h_l and/or t_l is present in e. Moreover e will be called the *information* of the string q.

With *input-axioms* we refer to the elemets of $C_A(A)$ and to their information; with *input-rules* we refer to the elements of $C_R(R)$ and to their information. The contexts will help the reader to understand in which meaning we use *input-axioms* and *input-rules*.

We will encode any arbitrary alphabet into the alphabet $W = \{0, 1, \hat{1}\}$, as follows. Assume that the input system has $V = \{v_1, v_2, ..., v_n\}, T = \{v_1, v_2, ..., v_m\}$ with $m \leq n$, then the input coding $f : V^* \to W^*$ is the morphism defined as

$$f(v) = \begin{cases} \hat{1}0^j \ \forall v = v_j \in V, j \leq m \\ 10^j \ \forall v = v_j \in V, m+1 \leq j \leq n \end{cases}$$

Now axioms are $C_A(x) = h_1 b\, f(x)\, s_1 t_1, \forall x \in A$.

An *input-axiom* is always in state 1 and the information it contains has the form: $b\langle code\rangle$, where $\langle code\rangle$ represents one axiom of the system to simulate. The

b indicates where the information of the axiom begins. This is important as *input-axioms* are rotated.

Let us define the set $\bar{W} = \{\bar{a} \mid a \in W\}$ and the homomorphism $bar : (W \cup \{\#, \$\})^* \to (\bar{W} \cup \{\bar{\#}, \bar{\$}\})^*$ by $bar\,(a) = \bar{a}$.

Moreover let the function *mirror image* be $mi : (W \cup \{\#\})^* \to (W \cup \{\#\})^*$ so that if $a = a_1...a_p$ is in $(W \cup \{\#\})^*$, then $mi(a) = a_p...a_1$.

If $x = u_1\#u_2\$u_3\#u_4$ is a rule present in the input system, then it will be added as axiom to the universal system, coded as follows:

$$C_R(x) = h_3h_2s_1bar(mi(f(u_1)\#f(u_2))\$mi(f(u_3)\#f(u_4)))t_2t_3$$

Observe that *input-axioms* and *input-rules* are written in opposite way. For example, if we consider the string $v_1v_1v_2v_3v_4$ and the splicing rule $v_1\#v_2v_3\$v_2\#v_3v_1$ their codings, according to the above, is:

$h_1b1010100100010000t_1$ and $h_3h_2s_1\bar{0}\bar{0}\bar{0}\bar{1}\bar{0}\bar{0}\bar{1}\bar{\#}\bar{0}\bar{1}\bar{\$}\bar{0}\bar{1}\bar{0}\bar{0}\bar{0}\bar{1}\bar{\#}\bar{0}\bar{0}\bar{1}t_2t_3$

Let us consider the splicing site $\bar{0}\bar{0}\bar{0}\bar{1}\bar{0}\bar{0}\bar{1}\bar{0}\bar{1}$ present on the left of $\bar{\$}$ in the *input-rule*. We may see that it is the mirror image with barred characters of the substring 101001000 present in the *input-axiom*. This is important for the operation of matching between a *input-rule* and a *input-axiom* performed by the system.

The decoding function D is defined as:

$$D(h_1vbws_1t_1) = f^{-1}(wv),$$

where f is the homomorphism for the encoding of the alphabets used above.

As the language generated by the universal system consists of strings in state 1, with a marker b indicating the start of the rotated string, the decoding is easily understood.

5 The Direct Universal System and Its Complexity

Theorem 1 *The system $\gamma_u = (V_u, T_u, A_u, R_u)$ of type (FIN, FIN) described in the previous section is universal for the class of EHSds: if the set of axioms $C_A(A) \cup C_R(R)$ related to a EHSds $\gamma = (V, T, A, R)$ is included in γ_u it will produce $L(\gamma_u(\gamma))$ so that $D(L(\gamma_u(\gamma))) = L(\gamma)$, where D is the decoding function defined in the previous section.*

Proof: The different paragraphs present in Section 3 indicate how γ_u can simulate a double splicing of the simulated system starting with the code of two axioms of that system. More detailed information about this process can be found in [7]. The two strings created at the end of the simulation of a double splicing have the same syntax used to code axioms of the simulated system. So they can be used to simulate subsequent splicings.

Now we are going to demonstrate that the only terminal strings created by γ_u are the ones coding strings generated by the simulated system.

Let us begin to say that none of the strings in A_u is terminal, so terminal strings can be present in the *input-axioms* and the *input-rules* or can be generated by splicing.

Our implementation of the algorithm with double splicing is made such that all rules used are in couples: the first one recognizes the pair (head, symbol) or (symbol, tail) in a string s and performs the first splicing with a short axiom belonging to A_u fully indicated in the splicing site of the used rule. In this way one end of s is changed and a new short string is created. This short string is particular as created by a specific pair (head, symbol) or (symbol, tail) of s and the axiom used in the first splicing. The second splicing rule of a couple has in one of its splicing site an essential part of the new created short string.

In this way it is possible to divide the set of rules of the universal system in two parts: the first composed by rules used for the first splicings, recognizing a pair (head, symbol) or (symbol, tail) present in a generic string s and using an axiom of γ_u, and the second composed by rules using the new short string generated in the previous step and the string s partially modified.

So no rule of the second set may be used as first because the short string they recognize cannot be present.

In this way we demonstrated that the evolution of a string is deterministic and it is driven by the pair (head, symbol) or (symbol, tail) present in a generic string s. If now we consider that only strings present in the *input-axioms* can contain the pairs (head, symbol) or (symbol, tail) priming the deterministic process, we end up to the conclusion that final strings can be obtained only starting from strings present in the *input-axioms*.

Short strings generated by a double splicing can be involved in another one as they can contain a pair (head, symbol) or (symbol, tail). But the operation will generate other non terminal short strings never involved in the deterministic process just described.

It is possible to reduce to one the number of axioms present in the universal system.

Theorem 2 *There exists a universal system for EHSds, which is of type* $([1], FIN)$.

Proof: Consider $\gamma_u = (V_u, T_u, A_u, R_u)$ the system defined in the previous section, and $\gamma = (V, T, A, R)$ element of the class of EHSds. We write $\gamma'_u = (V'_u, T_u, A'_u, R'_u)$ of type $([1], FIN)$, so that $D(L(\gamma'_u(\gamma))) = L(\gamma)\ \forall\gamma$ in EHSds. If we consider $V'_u = V_u \cup \{z\}$, where $z \notin V_u$, it is possible to write the single element of A'_u as $z\alpha_1 z\alpha_2 z \cdots \alpha_p z$ where $A_u \cup C_A(A) \cup C_R(R) = \{\alpha_1, \alpha_2, \cdots, \alpha_p\}$. Moreover $R'_u = R_u \cup \{\#z\$z\#\}$.

Applying twice the rule $\#z\$z\#$ to two occurrences of the axiom it is possible to obtain one element of A_u. Repeating this process all elements of A_u can be obtained so that γ'_u can elaborate as γ_u.

The rule $\#z\$z\#$ cannot be used on elements of A_u as $z \notin V_u$. On the other hand if splicing rules in R_u are applied to strings containing z they will not generate terminal strings as $z \notin T_u$, and this proves the theorem.

The number of splicing rules present in γ_u is 806, as listed in the Appendix of [7], and considering the repetition of rules to perform the second match and the second splicing. Even if still big, the number of rules is smaller than the number for a indirect universal system. If such a system is created using a universal

Turing machine, the minimal number of rules would be 222.220 (as indicated in Chapter 8 of [15]).

The complexity of the simulation of a simulated double splicing step can be estimated as follows. Let us consider that $x_1, x_2 \in (W \cup \{b\})^*$ are the information of the two strings involved in the simulation of the first splicing, and that $\mid x_1 \mid = p, \mid x_2 \mid = q$. Moreover let $y_1, y_2 \in \bar{W}^*$ be the code of the two splicing sites of the rule simulated on x_1 and x_2, $\mid y_1 \mid = r$ and $\mid y_2 \mid = s$. The symbol t and u are related to the number of symbols rotated during the ordering operation (paragraph 1.8) for the first splicing. The symbols p', q', r', s', t' and u' have the same meaning of p, q, r, s, t and u respectively for the second splicing.

The universal system will need
$10(p+q+p'+q')+4(rp+sq+r'p'+s'q')+t(4+4p)+u(4+4q)+t'(4+4p')+u'(4+4q')+48$
splicing operations to simulate a double splicing. The detailed explanation of the previous expression is present in [7].

6 Final Remarks

As expected the number of rules present in a direct universal H system is smaller than the one needed by an indirect one. The result described here is still far from a possible implementation in a biological laboratory, but we hope that this paper stimulates possible improvements of the algorithm. For instance, it is possible to consider to implement direct universal H systems using a control systems different from double splicing, or, for instance, it could be interesting to use circular strings. This last consideration comes from the fact that rotation, basic operation in our algorithm, is automatic in circular strings.

Moreover our algorithm is strongly linked to the idea of a sequential state machine: the pairs (head, symbol) or (symbol, tail) are really close to the way of working of Turing machines. Maybe it is possible to base a universal system on a totally different idea.

Considering that all extended H system with a control system generate the class or RE languages it is easy to imagine how to translate one system to another one through a Turing machine. The idea to simulate splicing using splicing that we described can be use to create direct translation between extended H systems with different control systems.

Acknowledgments

I thank the Universitá degli Studi di Milano for its financial support to my PhD, the Universiteit Leiden, personified by Prof. G. Rozenberg, accepting me as PhD student in his friendly group of research.

A special thank Gheorghe Păun and Joost Engelfriet for their stimulating ideas, and to Hendrik Jan Hoogeboom who read the draft copy of this paper and whose suggestions were really valuable for the final one.

References

1. G. Alford, Explicitly constructing universal extended H systems. In Calude et al. Proceedings of the UMC, Auckland, New Zealand, 5–11 January 1998
2. E. Csuhaj-Varju, R. Freund, L. Kari, Gh. Păun: DNA computing based on splicing: universality results. Proc. First Annual Pacific. Symposium in Biocomputing, Hawaii, 1996 (L. Hunter, T. E. Klein, eds.), World Sci., Singapore, 1996, 179 – 190
3. K. L. Denninghoff, R. W. Gatterdam: On the undecidability of splicing systems. *International Journal of Computer and Mathematics*, 27 (1989), 133 - 145
4. C. Ferretti, G. Mauri, S. Kobayashi, T. Yokomori: On the universality of Post and splicing systems. In Lawrence Hunter and Teri Klein, editors. *Biocomputing: Proceedings of the 1996 Pacific Symposium* pages 288–299. World Scientific Publishing Co., Singapore, January 1996, ISBN 981-02-2578-4.
5. R. Freund, F. Freund, M. Oswald: Universal H systems using multisets, man., 1997
6. R. Freund, L. Kari. G. Păun: DNA computing based on splicing. The existence of universal computers. Technical Report 185-2/FR-2/95, Technical Univ. Wien, 1995, and *Theories of Computer Sci.* in press
7. P. Frisco A direct construction of a universal extended H system, technical report 2000-08, Universiteit Leiden
8. P. Frisco, G. Mauri and C. Ferretti. Simulating Turing machines by extended mH systems. In G. Paun, editor, *Computing with bio-molecules. Theory and experiments*, pages 221–238. Springer Verlag, Berlin, Heidelberg, New York., 1998
9. T. Head, Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors, *Bulletin of Math. Biology*, 49 (1987), 737–759
10. A. M. Turing: On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, Ser. 2, 42 (1936), 230-265; a correction, 43 (1936), 544-546
11. Gh. Păun: DNA computing based on splicing: universality results. Proc. of the Sec. Int. Coll. on Universal Machines and Computations, Metz, 1998, Vol. I, 67-91
12. Gh. Păun, On the splicing operation, *Discrete Appl. Math.*, 70 (1996), 57–79
13. Gh. Păun: Splicing systems with targets are computationally complete. Inform. Processing Letters, 59 (1996), 129 - 133
14. Gh. Păun, G. Rozenberg, A. Salomaa: Computing by splicing. Programmed and evolving splicing systems. IEEE Inter. Conf. on Evolutionary Computing, Indianapolis, 1997, 273 - 277
15. G. Păun, G. Rozenberg, A. Salomaa. DNA Computing, Springer Verlag, 1998
16. Y. Rogozhin: Small universal Turing machines. *TCS*, 168-2 (1996), 215 - 240
17. C. E. Shannon: A universal Turing machine with two internal states. *Automata Studies, Annals of Mathematical Studies*, 34, Princeton Univ. Press, 1956, 157-165

Speeding-Up Cellular Automata by Alternations

Chuzo Iwamoto*, Katsuyuki Tateishi, Kenichi Morita, and Katsunobu Imai

Hiroshima University,
Higashi-Hiroshima, 739-8527 Japan
chuzo@hiroshima-u.ac.jp

Abstract. There are two simple models of cellular automata: a semi-infinite array (with left boundary) of cells with sequential input mode, called an iterative array (IA), and a finite array (delimited at both ends) of n cells with parallel input mode, called a bounded cellular array (BCA). This paper presents a quadratic speedup theorem for IAs and an exponential speedup theorem for BCAs by using alternations. It is shown that for any computable functions $s(n), t(n) \geq n$, every $s(n)t(n)$-time deterministic IA can be simulated by an $O(s(n))$-space $O(t(n))$-time alternating IA. Since any $t(n)$-time IA is $t(n)$-space bounded, every $(t(n))^2$-time deterministic IA can be simulated by an $O(t(n))$-time alternating IA. This leads to a separation result: There is a language which can be accepted by an alternating IA in $O(t(n))$ time but not by any deterministic IA in $O(t(n))$ time. It is also shown that every $t(n)$-time nondeterministic BCA can be simulated by a linear-time alternating BCA.

1 Introduction

One of the simplest models for parallel recognition of languages is the cellular automaton (CA). A CA is a one-dimensional array of identical finite-state automata, called cells, which are uniformly interconnected. Every cell operates synchronously at discrete time steps and changes its state depending on the previous states of itself and its neighbors. There are two simple models of CA: a semi-infinite array (with left boundary) of cells with sequential input mode, called an iterative array (IA), and a finite array (delimited at both ends) of n cells with parallel input mode, called a bounded cellular array (BCA).

There is a huge amount of literature on simulation and separation results between various types of CA. For example, Umeo et al. [19] showed that every real-time two-way BCA can be simulated by a linear-time one-way BCA (in which information is allowed to move one direction). Dyer [5] proved that every nondeterministic two-way BCA can be simulated by a nondeterministic one-way BCA. Ibarra et al. showed in [8] that linear-time one-way BCAs are equivalent to $2n$-time one-way IAs. Iwamoto et al. presented a time hierarchy theorem for IAs [11], but it is open whether a similar time hierarchy theorem holds for

* Corresponding author. This research was supported in part by Scientific Research Grant, Ministry of Education, Japan.

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 240–251, 2001.

BCAs. (More information on relationships among various models may be found in [2,3,4,9,10,18].)

In this paper, we investigate the relationship between alternating and (non)deterministic CA. We first present a quadratic speedup theorem for IAs. It is shown that for any computable functions $s(n), t(n) \geq n$, every $s(n)t(n)$-time deterministic IA can be simulated by an $O(s(n))$-space $O(t(n))$-time alternating IA. Since any $t(n)$-time IA is $t(n)$-space bounded, every $(t(n))^2$-time deterministic IA can be simulated by an $O(t(n))$-time alternating IA. It is known that if $t_1(n)$ and $t_2(n)$ are computable functions such that $\lim_{n\to\infty} \frac{t_1(n)}{t_2(n)} = 0$, then there is a language which can be accepted by a deterministic IA in $t_2(n)$ time but not by any deterministic IA in $t_1(n)$ time [11]. Thus, our speedup theorem leads to a separation result: There is a language which can be accepted by an alternating IA in $O(t(n))$ time but not by any deterministic IA in $O(t(n))$ time.

We also present an exponential speedup theorem for the fixed-space model. It is shown that for any computable function $t(n) \geq n$, every $t(n)$-time nondeterministic BCA can be simulated by a linear-time alternating BCA. (Note that $t(n) = 2^{O(n)}$ because the number of configurations of any BCA is bounded by $2^{O(n)}$.)

Computations on BCAs can be sped-up exponentially from $2^{O(n)}$ time to $O(n)$ time, while IAs can be sped-up quadratically. There is an essential difference between BCAs and IAs. Since an IA consists of an infinite array of cells, a $t(n)$-time IA can use $t(n)$ cellular space. Therefore, in order to speedup a $(t(n))^2$-time computation on an IA to $t(n)$ time, we must also compress cellular space from $(t(n))^2$ to $t(n)$. Our theorem for IAs is also a space-compression result.

The first speedup result for CA was shown by Smith [17]; he showed that $t(n)$-time BCAs can be sped-up to $n + t(n)/k$ time, where k is a constant. A linear speedup theorem for a synchronization problem was presented in [7]. Buchholz [1] considered arrays with restricted nondeterminism; it was shown that every linear-time computation on one-way one-guess BCAs can be sped-up to real-time. Ozhigov [15] investigated the trade-offs between the time and space complexities of computations on nondeterministic CA of dimension r. It was shown that a computation with time complexity $t(n)$ and space complexity $s(n)$ can be simulated with time and space complexity $O((t(n))^{1/(r+1)}(s(n))^{r/(r+1)})$.

Simulation results between alternating CA and alternating Turing machines (TMs) can be found in [12] and [13]. Krithivasan et al. [12] proved that alternating BCAs have the same power as linear-space alternating TMs. Matamala [13] proved that polynomial-time alternating CA have the same power as polynomial-time alternating TMs, and real-time alternating CA are strictly more powerful than real-time alternating TMs.

For TMs, several speedup results have been known. Wiedermann [20] showed that $\mathrm{NTIME}_1(t(n)) \subseteq \Sigma_2\text{-}\mathrm{TIME}_1(t(n)/\log t(n))$, where $\Sigma_2\text{-}\mathrm{TIME}_1(t(n))$ is the set of languages accepted by $t(n)$-time 1-tape TMs with one alternation from existential to universal. For multitape TMs, Gupta [6] proved that $\mathrm{DTIME}(t(n)) \subseteq \Sigma_2\text{-}\mathrm{TIME}(t(n)/\log^* t(n))$.

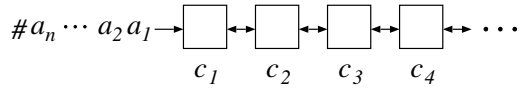


Fig. 1. Iterative array

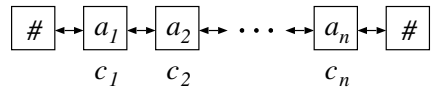


Fig. 2. Bounded cellular array

In the following section, we give the definition of CA. Main theorems are summarized in Section 3. The proofs are given in Sections 4 and 5.

2 Cellular Automata

A cellular automaton (CA) is a synchronous highly parallel string acceptor, consisting of a one-dimensional array of identical finite-state automata, called *cells*, which are uniformly interconnected. Every cell operates synchronously at discrete time steps and changes its state depending on the previous states of itself and its neighbors. There are two simple models of CA: a semi-infinite array (with left boundary) of cells with sequential input mode, called an *iterative array* (IA) (see Fig. 1), and a finite array (delimited at both ends) of n cells with parallel input mode, called a *bounded cellular array* (BCA) (see Fig. 2).

A nondeterministic IA is a 6-tuple $I = (Q, Q_a, \Sigma, \#, \delta, q)$, where

(1) Q is the finite nonempty set of cell states,
(2) Q_a is the accepting subset of Q,
(3) Σ is the finite input alphabet not in Q,
(4) $\#$ is the special boundary symbol not in $\Sigma \cup Q$,
(5) $\delta : Q \cup \Sigma \cup \{\#\} \times Q \times Q \to 2^Q$ is the local transition function,
(6) $q \in Q$ is the quiescent state such that $\delta(q, q, q) = \{q\}$ and if $\delta(a, b, c) \ni q$ then $b = q$.

The cell assigned to the integer $i \geq 1$ is denoted by c_i. At step $t = 0$, the state of each cell is the quiescent state q. The input string $a_1 a_2 \cdots a_n$, where $a_i \in \Sigma$, is fed serially to the leftmost cell c_1; the symbol a_i, $1 \leq i \leq n$, is the left neighbor of the cell c_1 at step $i-1$. After step $n-1$, the left neighbor of c_1 is the boundary symbol $\#$. A configuration of an IA is represented by a string in $\Sigma^*(Q - \{q\})^*$.

A nondeterministic BCA is a 5-tuple $B = (Q, Q_a, \Sigma, \#, \delta)$, where

(1) Q is the finite nonempty set of cell states,
(2) Q_a is the accepting subset of Q,
(3′) Σ is the finite input alphabet in Q,
(4′) $\#$ is the special boundary state not in Q,
(5′) $\delta : Q \cup \{\#\} \times Q \times Q \cup \{\#\} \to 2^Q$ is the local transition function.

The input string $a_1a_2\cdots a_n$ is applied to the array in parallel at step 0 by setting the states of cells $c_1, c_2, \ldots, c_n$ to $a_1, a_2, \ldots, a_n$, respectively. A configuration of a BCA is represented by a string in Q^n.

The definition of a computation tree is mostly from [16]. Computations of a nondeterministic CA given some input are described as a tree T: All nodes are configurations, the root is the initial configuration of the CA for the given input, and the children of a configuration C are exactly those configurations which can be reached from C in one step allowed by the transition function. Leaves of the tree T are final configurations, they may be accepting or rejecting. Certain paths in T may be infinite.

An interior node of the tree T is defined to be accepting if at least one of its children is accepting; the CA accepts the input iff the root is accepting. A nondeterministic CA is said to be $t(n)$-*time bounded* if, for every accepted input of length n, there is an accepting node whose distance from the root is at most $t(n)$.

The states of an alternating CA are partitioned into four classes: accepting states, rejecting states, universal states, and existential states. Whether a particular configuration is a universal, an existential, an accepting, or a rejecting configuration is determined by the state of the leftmost cell c_1. (This model is called a *week* alternating CA in [13].)

An existential interior node is defined to be accepting if at least one of its children is accepting. A universal interior node is defined to be accepting if all its children are accepting. An alternating CA M is defined to be $t(n)$-*time bounded* if, for every accepted input w of length n, the computation tree T of M started with w stays accepting if it is pruned at depth $t(n)$. A subtree of T is called a rejecting subtree if its root is not accepting. An alternating CA M is defined to be $s(n)$-*space bounded* if, for every accepted input w of length n, every node of the subgraph obtained from T by removing all rejecting subtrees is represented by a string of length at most $s(n)$.

Clearly, alternating CA without universal states are just nondeterministic CA. Nondeterministic CA of which each node of the computation tree has at most one child are deterministic CA.

3 Speedup Theorems

It is well known that linear-time speedup and linear-space compression theorems hold for CA [17]. In this paper, we present a quadratic speedup theorem for IAs and an exponential speedup theorem for BCAs by using alternations. Let $bin(n)$ denote the binary string (of length $\lceil \log n \rceil$) which represents the value n.

Theorem 1. *Suppose that $s(n) \geq n$ and $t(n) \geq n$ are computable functions such that there is an $O(s(n))$-space $O(t(n))$-time TM which, given $bin(n)$, generates both $bin(s(n))$ and $bin(t(n))$. For any $s(n)t(n)$-time deterministic IA M, there exists an $O(s(n))$-space $O(t(n))$-time alternating IA which can simulate M.*

The proof of this theorem is given in Section 4. (The definition of computable functions are from [11]. Note that the set of computable functions $t(n)$

and $s(n)$ includes all functions computable by polynomial-time TMs, since the input length of TMs is $\lceil \log n \rceil$.) Since $s(n)t(n)$-time IAs are obviously $s(n)t(n)$-space bounded, "$s(n)t(n)$-space" is omitted in the statement of Theorem 1. If we omit "$O(s(n))$-space" from Theorem 1, we obtain the following corollary.

Corollary 1. *Suppose that $t(n) \geq n$ is a computable function such that there is an $O(t(n))$-time TM which, given bin(n), generates bin$(t(n))$. For any $(t(n))^2$-time deterministic IA M, there exists an $O(t(n))$-time alternating IA which can simulate M.*

It is known that if $t_1(n)$ and $t_2(n)$ are computable functions such that $\lim_{n\to\infty} \frac{t_1(n)}{t_2(n)} = 0$, then there is a language which can be accepted by a deterministic IA in $t_2(n)$ time but not by any deterministic IA in $t_1(n)$ time [11]. Therefore:

Corollary 2. *Suppose that $t(n) \geq n$ is a computable function such that there is an $O(t(n))$-time TM which, given bin(n), generates bin$(t(n))$. Then, there exists a language which can be accepted by an alternating IA in $O(t(n))$ time but not by any deterministic IA in $O(t(n))$ time.*

We turn our attention to time complexity of the fixed-space model.

Theorem 2. *Suppose that $t(n) \geq n$ is a computable function such that there is an $O(n)$-time TM which, given bin(n), generates bin$(\lceil \log t(n) \rceil)$. For any $t(n)$-time nondeterministic BCA M, there exists a linear-time alternating BCA which can simulate M.*

The proof is given in Section 5. Note that the number of configurations of any BCA is bounded by $2^{O(n)}$. Theorem 2(ii) implies that nondeterministic BCAs can be sped-up exponentially.

4 Quadratic Speedup Algorithm for IAs

In this section, we prove Theorem 1. Let M_D be an arbitrary $s(n)t(n)$-time deterministic IA. We construct an $O(s(n))$-space $O(t(n))$-time alternating IA, say, M_A, which can simulate M_D. A function $f(n)$ is said to be *constructible* if, for each n, there is an IA whose accepting cell c_1 enters an accepting state at step $f(n)$ on all inputs of length n. It is known that all functions computable by TMs are constructible by IAs [11]. Since $s(n)$ and $t(n)$ are computable functions, we can assume they are also constructible.

Before describing details, we observe some basic ideas of the proof using a simple example. Consider the ith cell of M_D (see Fig. 3). The state of the ith cell at step τ is determined by the states of $2s(n)+1$ cells at step $\tau - s(n)$. In order to describe such a situation, the array of M_A's cells is divided into *tracks* (see Fig. 4). The second track contains the state of the ith cell of M_D at step τ. In order to simulate an $s(n)$-step computation of M_D, M_A existentially generates (guesses) $2s(n)+1$ states in the first track. Then, using a universal branch (see

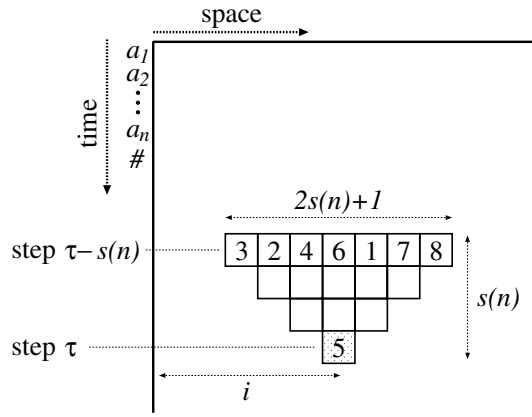


Fig. 3. Time-space diagram of deterministic IA M_D

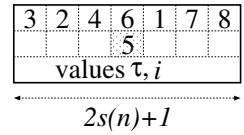


Fig. 4. Tracks of alternating IA M_A

Fig. 5(b)), M_A verifies whether the generated $2s(n) + 1$ states are consistent with the state of the ith cell of M_D at step τ according to the transition function of M_D. Also, M_A goes up to the next level (see Fig 5(a)); M_A universally chooses one cell in order to start the simulation of the next level (see the cell having state 7 in Fig. 5). Then, M_A stores the state 7 of the chosen cell into the second track in $O(1)$ steps. Furthermore, M_A computes the values $\tau - s(n), i + j$ and stores them into the third track in $O(1)$ steps. (These $O(1)$-step procedures require some techniques, which will be described later.) M_A repeats this algorithm $t(n)$ times. Finally, M_A verifies whether the first track contains a part of the initial configuration of M_D (see track G in Fig. 6). In the following, we describe the algorithm in detail.

(1) At step 1, M_A performs three procedures (1.1), (1.2) and (1.3) in parallel using three tracks. (1.1) M_A stores the input string $a_1a_2 \cdots a_n\#$ into n cells, $c_1, c_2, \cdots, c_n$, where $a_n\#$ is stored into c_n as a single symbol $a_n^{\#}$. This procedure needs only $2n$ steps. (1.2) M_A simulates an IA, say, M_{4s+2}, which makes exactly $4s(n) + 2$ moves. (1.3) M_A sends a unit-speed pulse from c_1. On the pulse, M_A existentially generates (guesses) a string of the form $00 \cdots 01$ of length $2s(n)+1$.

(2) M_A universally performs procedures (2.1) and (2.2) simultaneously.

(2.1) M_A verifies whether the string $00 \cdots 01$ has length $2s(n) + 1$ as follows. The cell containing "right-boundary" symbol 1 sends a unit-speed pulse

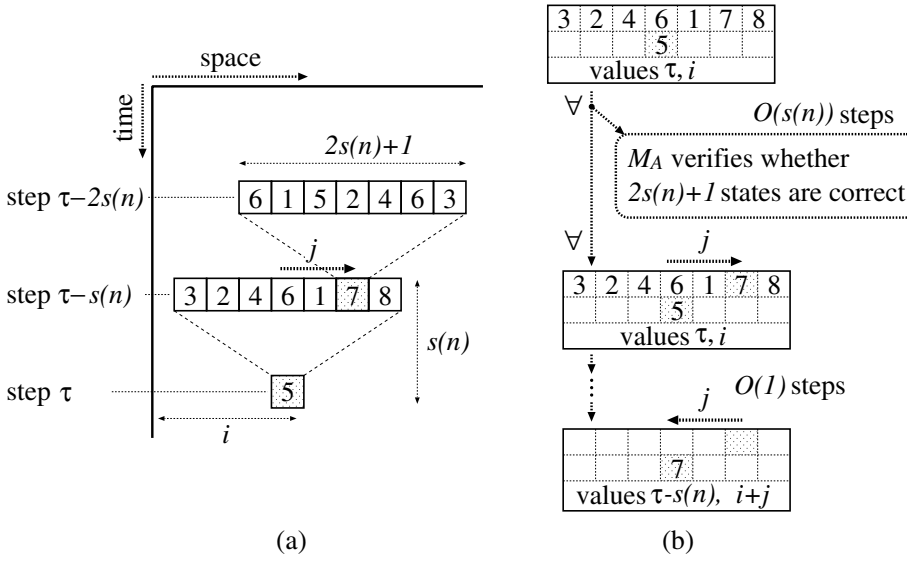


Fig. 5. $s(n)$-step computation of M_D and simulation of M_D by M_A

toward c_1. If the pulse reaches c_1 and M_{4s+2} enters an accepting state at the same time, then M_A knows that the length of $00\cdots01$ is exactly $2s(n)+1$, and M_A halts with an accepting state; otherwise, M_A halts with a rejecting state.

(2.2) M_A assumes that the region filled by the string $00\cdots01$ has length $2s(n)+1$. In the following, M_A does not violate the space limitation $2s(n)+1$. The array of M_A's cells is divided into tracks. In a track, say, A, M_A copies the input string $a_1a_2\cdots a_n^{\#}$ as many as possible in order that any $2n-1$ contiguous cells contain the input string (see Fig. 6). This can be done in $O(s(n))$ steps.

M_A uses tracks G, S, T, I, and C. (G, S, T, I, and C stand for *Guessed states, current State, step* τ*, cell* i, and *Counter*. Three tracks in Fig. 4 correspond to track G, track S, and tracks T, I.) Each of tracks T and I is further divided into subtracks. Let α_i, g_i, s_i be subcells of cell c_i in tracks A, G, S, respectively. We denote the state of the ith cell of M_D at step τ by $state(\tau, i)$. Note that $state(\tau, 0) = a_\tau$ for $0 \le \tau \le n-1$, and $state(\tau, 0) = \#$ for $\tau \ge n$. For convenience, we assume $state(\tau, i) = \#$ for $i < 0$. In order that all cells of M_A start the following procedure (3) simultaneously, M_A uses the firing squad synchronization algorithm [14].

(3) Without loss of generality, we can assume that M_D has the unique accepting state q_a. Initially, track S contains a string $q_aq_a\cdots q_a$ of length $2s(n)+1$. Note that all subcells (including the center subcell) of track S contain the same state q_a. Tracks T and I contain values $\tau = s(n)t(n)$ and $i = 1$, respectively. It should be noted that $t(n) \ge s(n)$, and the values of computable functions $s(n)$ and $t(n)$ are represented by binary strings of length $\lceil \log t(n) \rceil$. Thus, the value

of $s(n)t(n)$ can be computed in time polynomial in $\log t(n)$. Tracks T, I, S represent that the leftmost cell of M_D has state q_a at step $\tau = s(n)t(n)$ (i.e., $state(\tau, 1) = q_a$).

Note that $state(\tau, i)$ depends on $2s(n)+1$ states at step $\tau - s(n)$ as follows:

$$state(\tau - s(n), i - s(n)), \ldots, state(\tau - s(n), i), \ldots, state(\tau - s(n), i + s(n))$$

Track G contains these $2s(n)+1$ states, which are guessed existentially. Note that these states can be generated in parallel within $O(1)$ steps. Furthermore, in a subtrack of T, M_A existentially guesses the new value $\widehat{\tau} = \tau - s(n)$ as a binary string, which can also be generated in $O(1)$ steps.

(4) M_A performs the following procedures (4.1), (4.2), and (4.3) universally.

(4.1) M_A verifies whether the guessed value $\widehat{\tau}$ is equal to $\tau - s(n)$. Since τ and $s(n)$ can be represented by binary strings of length $O(\log t(n))$, the value of $\tau - s(n)$ can be computed in time polynomial in $\log t(n)$. If $\widehat{\tau} = \tau - s(n)$, then M_A halts with an accepting state; otherwise M_A halts with a rejecting state.

(4.2) M_A verifies whether the above $2s(n)+1$ states are consistent with the state in track S by making an $s(n)$-step simulation of M_D. If they are consistent with the state in track S according to the transition function of M_D, then M_A halts with an accepting state; otherwise M_A halts with a rejecting state.

(4.3) M_A assumes that $\widehat{\tau} = \tau - s(n)$ and the $2s(n)+1$ states are consistent with the state in track S. M_A universally chooses one of the $2s(n)+1$ cells of track G as follows. (The state, say, q', of the chosen cell will be the new state in track S. See the state 7 in Fig. 5(b)) M_A universally generates a value $\widehat{j}$ (where $-s(n) \leq \widehat{j} \leq s(n)$) as a binary string in a subtrack of I within $O(1)$ steps. Then, M_A existentially generates a string of the form $0^{s(n)+j}10^{s(n)-j}$ of length $2s(n)+1$, where $-s(n) \leq j \leq s(n)$. The following (4.3.1) and (4.3.2) are universally performed.

(4.3.1) M_A verifies whether (i) exactly one 1 appears in the generated string of length $2s(n)+1$ and (ii) this 1 appears at the $(s(n)+\widehat{j}+1)$st position (i.e., $\widehat{j} = j$). (i) can be verified by a single scan. In order to verify (ii), the symbol 1 sends a unit-speed pulse toward the leftmost cell, and M_A decreases the value one by one from $s(n)+\widehat{j}+1$ to 0 (using the counting algorithm in [11]), doing both simultaneously. If the pulse reaches the leftmost cell and the value becomes 0 at the same time, M_A halts with an accepting state; otherwise, M_A halts with a rejecting state. (Note that the value of $s(n)+\widehat{j}+1$ can be computed in time polynomial in $\log t(n)$.)

(4.3.2) M_A stores the state q' of the chosen cell into a subcell of track S. In order to generate a string $q'q' \cdots q'$ of length $2s(n)+1$, M_A existentially generates (guesses) an arbitrary string $\sigma \in {Q_D}^*$ of length $2s(n)+1$, where Q_D is the state set of M_D. M_A performs (4.3.2a) and (4.3.2b) universally.

(4.3.2a) Every subcell of track S verifies (locally) whether the generated string is over one-letter alphabet, and the subcell at the $(s(n)+\widehat{j}+1)$st position verifies whether the letter is q'. M_A halts with a rejecting state if $\sigma \neq q'q' \cdots q'$; otherwise, M_A halts with an accepting state. (The rejecting pulse reaches the leftmost cell in $O(s(n))$ steps.)

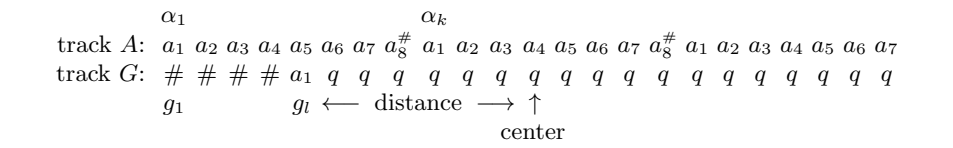


Fig. 6. Tracks A and G

(4.3.2b) M_A assumes that $\sigma = q'q' \cdots q'$, $\widehat{\tau} = \tau - s(n)$, and $\widehat{j}$ is correct. M_A regards $\widehat{\tau}$ and $i + \widehat{j}$ as τ and i, respectively. As in (3), in track G, M_A existentially generates $2s(n) + 1$ states in $O(1)$ steps. Then, in a subtrack of T, M_A existentially generates the new value $\widehat{\tau} = \tau - s(n)$ in $O(1)$ steps. Again, M_A performs the procedure (4).

During the above procedures, M_A simulates an IA which makes exactly $t(n)$ moves in track C. (Recall that $t(n)$ is a constructible function.) At the $t(n)$th recursive call of the procedure (4), M_A verifies whether $2s(n)+1$ states in track G are consistent with initial configuration of M_D (see track G in Fig. 6), i.e.,

$$\cdots\cdots, state(0,-1) = \#, state(0,0) = a_1, state(0,1) = q, state(0,2) = q, \cdots\cdots,$$

where $a_1 \in \Sigma$ is the first symbol of the input string, # is the boundary symbol, and q is the quiescent state. This verification is described as follows: First of all, M_A verifies whether the sequence of $2s(n) + 1$ states in track G is a string in set $\{\#\}^* \cup \{\#\}^*\Sigma\{q\}^* \cup \{q\}^*$. This can be verified by a single scan. Suppose that a subcell, say, g_l, of track G contains $a_1 \in \Sigma$.

Let k be the smallest integer such that $l \leq k$ and subcell α_k in track A contains a_1. In order to make an $s(n)$-step simulation of M_D, the subcell g_l needs the input string $a_1 a_2 \cdots a_n^{\#}$. Thus, M_A must shift the sequence in track A $k - l$ positions to the left as follows.

The leftmost subcells g_1 and α_1 send two pulses p_g and p_α, respectively. When the pulse p_g (resp. p_α) reaches g_l (α_k), the subcell g_l (α_k) sends a pulse p'_g (p'_α) toward g_1 (α_1). Therefore, at steps $2l$ and $2k$, these pulses reach g_1 and α_1, respectively. M_A starts the firing squad synchronization algorithm at steps $2l$ and $2k$. During the time period from the first synchronization to the second synchronization, M_A can shifts the sequence in track A $k - l$ positions to the left. Also, M_A verifies whether the distance from g_l to the center of the $2s(n) + 1$ cells is equal to the value i in track I. This can be verified by using a unit-speed pulse from g_l toward the center cell and the counting algorithm in [11]. The time complexity of this procedure is bounded by $O(s(n))$.

The procedure in the previous paragraph and procedures (1.1), (1.2), (2.1), (4.1), (4.2), (4.3.1) and (4.3.2a) are "leaves" of the computation tree of M_A. All these procedures are performed in at most $O(t(n))$ steps. (Note that $s(n) \leq t(n)$.) Procedures (1.3), (2.2) and (3) are performed once; their time complexities are $O(t(n))$. Each of the remaining procedures can be executed in $O(1)$ steps. Procedure (4) is executed $t(n)$ times. Therefore, the time complexity of M_A is bounded by $O(t(n))$. Since each execution of (4) simulates an $s(n)$-step computation

of M_D, alternating IA M_A can simulate an $s(n)t(n)$-step computation of M_D. As we mentioned in (2.2), the space complexity of M_A is bounded by $O(s(n))$. This completes the proof of Theorem 1.

5 Exponential Speedup Algorithm for BCAs

In this section, we prove Theorem 2. Let M_N be an arbitrary $t(n)$-time nondeterministic BCA. We construct a linear-time alternating BCA, say, M_A, which can simulate M_N. It should be noted that the value $t(n)$ is bounded by $2^{O(n)}$ (and thus $\log t(n) = O(n)$), since the number of configurations of any BCA is bounded by $2^{O(n)}$.

First of all, we give a sketch of the proof. M_A verifies whether there is a $t(n)$-step computation path of M_N from the initial configuration to a final configuration as follows. M_A existentially generates the configuration of M_N at step $t(n)/2$. Then, using a universal branch, M_A verifies whether both the first half computation and the second half computation are correct. M_A repeats this procedure $\lceil \log t(n) \rceil$ times recursively (see below for the details). In order to execute this algorithm, M_A must compute the value of $\lceil \log t(n) \rceil$ in advance. Furthermore, computing the value $\lceil \log t(n) \rceil$ requires the value of n. For these reasons, M_A performs the procedure given in the following two paragraphs.

(1) Using the same technique in [11], M_A generates binary string $bin(n)$, which represents the value n. This can be done in $O(n)$ steps. M_A computes the value of $\lceil \log t(n) \rceil$ using the generated value n; M_A simulates a TM which, given $bin(n)$, generates $bin(\lceil \log t(n) \rceil)$ in time $O(n)$ and space $O(n)$. Note that the length of the input of the TM is $\lceil \log n \rceil$. Using the linear-speedup theorem [17], this simulation can be done in n steps and n cells.

(2) M_A uses tracks, say, U, V, W and X. For $1 \leq i \leq n$, let u_i, v_i, w_i and x_i be subcells of c_i in tracks U, V, W and X, respectively. We denote the configuration of M_N at step τ by $C(\tau)$. We consider three integers u, v, w satisfying $u = 0$, $v = \lfloor (u + w)/2 \rfloor$, and $w = t(n)$. (Note that M_A does not have to compute these values.)

Initially, track U contains the input string (i.e., initial configuration $C(0)$ of M_N). Subcell w_1 contains an accepting state q_a of M_N and the remaining subcells $w_2, \ldots, w_n$ contains $n-1$ states, which are guessed existentially. These n states can be generated in one step in parallel. (Subcells $w_1, w_2, \ldots, w_n$ correspond to an accepting configuration $C(t(n))$.) Also, subcells $v_1, v_2, \ldots, v_n$ contains n states which are guessed existentially. (These states correspond to $C(v)$.)

(3) The leftmost cell universally generates a symbol $s \in \{0, 1\}$. Then, M_A existentially generates a string $\sigma \in \{0, 1\}^*$ of length n in track X. M_A performs the following procedures (3.1) and (3.2) universally.

(3.1) M_A verifies whether σ is over one-letter alphabet, and the leftmost cell verifies whether the letter is s. M_A halts with a rejecting state if $\sigma \neq ss \cdots s$; otherwise M_A halts with an accepting state. (The rejecting pulse reaches the leftmost cell in n steps.)

(3.2) Under the assumption $\sigma = ss\cdots s$, M_A stores the state of every subcell v_i into w_i (resp. u_i) if x_i contains $s = 0$ (resp. $s = 1$). (Consider updated integers $u := u$, $w := v$, and $v := \lfloor (u + w)/2 \rfloor$ (resp. $u := v$, $w := w$, and $v := \lfloor (u + w)/2 \rfloor$).) M_A existentially generates n states in parallel in one step, which are stored into subcells $v_1, v_2, \ldots, v_n$. (These states correspond to configuration $C(v)$). Again, M_A performs the procedure (3).

During the above procedures, M_A counts the number of calls of (3) by decreasing the value one by one from $\lceil \log t(n) \rceil$ to 0. When the value becomes 0, M_A changes the configuration in track U into a one-step-after configuration according to the transition function of M_N. If the one-step-after configuration is equal to the configuration in track W, then M_A halts with an accepting state; otherwise M_A halts a rejecting state. (The accepting or rejecting pulse reaches the leftmost cell in n steps.)

The procedure in the previous paragraph and procedure (3.1) are "leaves" of the computation tree of M_A, which can be done in $O(n)$ steps. Procedures (1) and (2) can be performed once; the time complexity of them is $O(n)$. Each of the remaining procedures can be performed in $O(1)$ steps. Procedure (3) is executed $\lceil \log t(n) \rceil$ times. Since $\log t(n) = O(n)$, the time complexity of M_A is bounded by $O(n)$. This completes the proof of Theorem 2.

References

1. T. Buchholz, A. Klein, and M. Kutrib, One guess one-way cellular array, in: *Proc. MFCS (LNCS1450)*, 1998, 807–815.
2. J.H. Chang, O.H. Ibarra, and A. Vergis, On the power of one-way communication, *J. ACM*, **35** 3 (1988) 697–726.
3. C. Choffrut and K. Culik II, On real-time cellular automata and trellis automata, *Acta Informatica*, **21** (1984) 393–407.
4. S.N. Cole, Real-time computation by n dimensional iterative arrays of finite-state machines, *IEEE Trans. on Computers*, **C-18** 4 (1969) 349–365.
5. C.R. Dyer, One-way bounded cellular automata, *Inform. and Control*, **44** (1980) 261–281.
6. S. Gupta, Alternating time versus deterministic time: a separation, in: *Proc. Structure in Complexity*, 1993, 266–277.
7. O. Heen, Linear speed-up for cellular automata synchronizers and applications, *Theoret. Comput. Sci.*, **188** (1997) 45–57.
8. O.H. Ibarra and T. Jiang, On one-way cellular arrays, *SIAM J. Comput.*, **16** 6 (1987) 1135–1154.
9. O.H. Ibarra and T. Jiang, Relating the power of cellular arrays to their closure properties, *Theoret. Comput. Sci.*, **57** (1988) 225–235.
10. O.H. Ibarra and M.A. Palis, Two-dimensional iterative arrays: characterizations and applications, *Theoret. Comput. Sci.*, **57** (1988) 47–86.
11. C. Iwamoto, T. Hatsuyama, K. Morita, and K. Imai, On time-constructible functions in one-dimensional cellular automata, in: *Proc. Int'l Symp. on Fundamentals of Computation Theory (LNCS1684)*, 1999, 316–326.
12. K. Krithivasan and M. Mahajan, Nondeterministic, probabilistic and alternating computations on cellular array models, *Theoret. Comput. Sci.*, **143** (1995) 23–49.

13. M. Matamala, Alternation on cellular automata, *Theoret. Comput. Sci.*, **180** (1997) 229–241.
14. J. Mazoyer, A 6-state minimal time solution to the firing squad synchronization problem, *Theoret. Comput. Sci.*, **50** (1987) 183–238.
15. Y. Ozhigov, Computations on nondeterministic cellular automata, *Inform. and Computation*, **148** (1999) 181–201.
16. W.J. Paul, E.J. Prauß, and R. Reischuk, On alternation, *Acta Informatica* **14** (1980) 243–255.
17. A.R. Smith III, Real-time recognition by one-dimensional cellular automata, *J. of Comput. and System Sci.*, **6** (1972) 233–253.
18. V. Terrier, On real time one-way cellular array, *Theoret. Comput. Sci.*, **141** (1995) 331–335.
19. H. Umeo, K. Morita, and K. Sugata, Deterministic one-way simulation of two-way real-time cellular automata and its related problems, *Inform. Process. Lett.*, **14** 4 (1982) 158–161.
20. J. Wiedermann, Speeding-up single-tape nondeterministic computations by single alternation, with separation results, in: *Proc. MFCS (LNCS1099)*, 1996, 381–392.

Efficient Universal Pushdown Cellular Automata and Their Application to Complexity

Martin Kutrib

Institute of Informatics, University of Giessen
Arndtstr. 2, D-35392 Giessen, Germany
kutrib@informatik.uni-giessen.de

Abstract. In order to obtain universal classical cellular automata infinite space is required. Therefore, the number of required processors depends on the length of input data and, additionally, may increase during the computation. On the other hand, Turing machines are universal devices which have one processor only and additionally an infinite storage tape. Here an in some sense intermediate model is studied. The pushdown cellular automata are a stack augmented generalization of classical cellular automata. They form a massively parallel universal model where the number of processors is bounded by the length of input data.
Efficient universal pushdown cellular automata and their efficiently verifiable encodings are proposed. They are applied to computational complexity, and tight time and stack-space hierarchies are shown.

1 Introduction

Arrays of automata can be understood as models for massively parallel computers. By treating them as acceptors for formal languages their computational power can be compared with other parallel and sequential computer models. Under these aspects the automata arrays and various modifications have been studied for a long time. Especially investigations concerning universality (often combined with other properties) have been done e.g. in [1,5,11,12,14,15]. A state-of-the-art survey on universality and decidability versus undecidability in cellular automata and several other models of discrete computations can be found in [10].

Due to the historical precedent for a fixed amount of memory per cell (unbounded) cellular automata have to be defined over an infinite space in order to obtain computational universality. Therefore, the number of required processors depends on the length of input data and, additionally, may increase during the computation. From a more practical point of view an infinite number of processors seems to be fairly unrealistic. On the other hand Turing acceptors are computationally universal devices which have one processor only and additionally an infinite storage tape. For this reason and due to the possible speed-up gained in parallelism we investigate the pushdown cellular automata (PDCA) where each cell is now a deterministic pushdown automaton [7,9]. So we obtain a computationally universal computer model where the number of processors is

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 252–263, 2001.

bounded by the length of input data. Furthermore, in our opinion the assumption of arbitrarily large pushdown memory is less problematical than the assumption of an arbitrarily number of processors.

Clearly, a PDCA with at least two cells is sufficient in order to obtain an universal device. But with an eye towards applications e.g. in diagonalization proofs here we are interested in efficient universal PDCAs.

2 Basic Notions

We denote the integers by $\mathbb{Z}$, the positive integers $\{1, 2, ...\}$ by $\mathbb{N}$ and the set $\mathbb{N} \cup \{0\}$ by $\mathbb{N}_0$. The empty word is denoted by λ and the reversal of a word w by w^R. For the length of w we write $|w|$. We use $\subseteq$ for inclusions and $\subset$ if the inclusion is strict. $proj_i(x_1 \cdots x_n) = x_i$ selects the ith component of a word or a vector.

A pushdown cellular automaton is a linear array of identical deterministic pushdown automata, sometimes called cells, where each of them is connected to its both nearest neighbors (one to the right and one to the left). For convenience we identify the cells by positive integers. They operate synchronously at discrete time steps. The state transition of a cell depends on the current states of its both neighbors, the current state of the cell itself and the current symbol at the top of its stack. With an eye towards language recognition we provide accepting and rejecting states. More formally:

Definition 1. *A* pushdown cellular automaton *(PDCA) is a system* $\langle S, G, \delta_s, \delta_p, \#, \bot, A, F_+, F_- \rangle$*, where*

1. S *is the finite, nonempty set of* states,
2. G *is the finite, nonempty set of* stack symbols,
3. $\# \notin S$ *is the* boundary state,
4. $\bot \in G$ *is the* bottom-of-stack symbol,
5. $A \subseteq S$ *is the finite, nonempty set of* input symbols,
6. F_+ *and* F_-, $F_+ \cap F_- = \emptyset$, *are the sets of* accepting *and* rejecting *states, respectively,*
7. $\delta_s : (S \cup \{\#\})^3 \times G \to S$ *is the* local state transition function,
8. $\delta_p : (S \cup \{\#\})^3 \times G \to \{\lambda\} \cup G \cup G^2$ *is the* local stack transition function *satisfying* $\forall s_1, s_2, s_3 \in S, g \in G \setminus \{\bot\}$:

$$\delta_p(s_1, s_2, s_3, \bot) \in \left\{ g'\bot \mid g' \in (G \setminus \{\bot\}) \cup \{\lambda\} \right\} \text{ and}$$
$$\delta_p(s_1, s_2, s_3, g) \in \{\lambda\} \cup (G \setminus \{\bot\}) \cup (G \setminus \{\bot\})^2.$$

The condition on the local stack transition function ensures that the bottom-of-stack symbol appears at each cell exactly once (i.e. at the bottom of its stack). At every transition step each cell consumes the symbol at the top of its stack (if it is not empty) and pushes at most two new symbols onto it. Note that the restriction of pushing at most two symbols at every time step neither reduces the computation power nor slows down the computation itself [4].

Let $\mathcal{M} = \langle S, G, \delta_s, \delta_p, \#, \bot, A, F_+, F_- \rangle$ be a PDCA. A configuration of $\mathcal{M}$ at some time $t \geq 0$ is a mapping $c_t : [1, \ldots, n] \to S \times G^+$ for $n \in \mathbb{N}$, that maps the cells to their current states and stack contents. The configuration at time 0 is defined by the input symbols and empty stacks. For a given input $w = a_1 \cdots a_n \in A^+$ we set $c_0(i) = (a_i, \bot)$,$1 \leq i \leq n$. Subsequent configurations are computed by synchronously applying both local transition functions to all cells in parallel. The virtual left neighbor of the leftmost cell and the virtual right neighbor of the rightmost cell are always assumed to be in the boundary state.

3 Encoding of Pushdown Cellular Automata

Needless to say, in general it is possible to encode PDCAs and their configurations with any nonempty alphabet. But with an eye towards applications in formal language recognition, we are interested in efficiently verifiable encodings that have to be chosen with respect to the processing universal PDCA. Later on, the encodings in combination with the universal PDCA will determine the tightness of the hierarchies of complexity classes.

Let $\text{bin} : \mathbb{N}_0 \to \{0,1\}^+$ be the mapping that maps a natural number to its binary representation without leading zeroes. Then the binary representation with leading zeroes is for all $k \geq 1$ defined by $\text{bin}_k : \mathbb{N}_0 \to \{0,1\}^+$, $n \mapsto 0^{k-|\text{bin}(n)|}\text{bin}(n)$ if $k \geq |\text{bin}(n)|$. $\text{bin}_k(n)$ is undefined for $k < |\text{bin}(n)|$.

S and G are finite nonempty sets and we can assume total orderings on their elements: $S = \{s_1, \ldots, s_{|S|}\}$ and $G = \{g_1, \ldots, g_{|G|}\}$. W.l.o.g. let s_0 denote the boundary state # and g_1 be the bottom-of-stack symbol $\bot$. The different beginnings of the numberings have been chosen since for stack operations the empty word λ has to be encoded in addition. The ordering of the state set implies orderings of F_+ and F_-.

The state and stack transition functions can be represented as a table with seven columns. Each row contains the states of the left neighbor, of the cell itself and of the right neighbor and the symbol at the top of the stack, followed by the next state and the two symbols (which may be λ) that form the new top of the stack:

$$s_i \quad s_j \quad s_k \quad g_l \quad \delta_s(s_i, s_j, s_k, g_l) \quad proj_1(\delta_p(s_i, s_j, s_k, g_l)) \quad proj_2(\delta_p(s_i, s_j, s_k, g_l))$$

We assume that the rows are in lexicographic order. If for certain s_i, s_j, s_k, g_l the corresponding row is missing then $\delta_s(s_i, s_j, s_k, g_l)$ is defined to be s_j and $\delta_p(s_i, s_j, s_k, g_l)$ is defined to be g_l (i.e. neither the state nor the stack content changes).

Let $k = |\text{bin}(\max\{|G|, |S|\})|$ and $C = \{\texttt{0}, \texttt{1}, \texttt{[}, \texttt{]}, \texttt{+}, \texttt{-}, \texttt{b}, \lambda\}$.

1. The states, stack symbols and the empty word are encoded by a pair of square brackets that contain the corresponding binary representation with leading zeroes followed by a mark (+, -, b) that indicates that the coded

symbol belongs to F_+, F_- or not at all. Thus, we obtain a mapping as follows:

$$code_S : S \cup \{\#\} \cup G \cup \{\lambda\} \to C^{k+3}$$

$$p \mapsto \begin{cases} [\mathrm{bin}_k(i)+] & \text{if } p = s_i \in S \wedge s_i \in F_+ \\ [\mathrm{bin}_k(i)-] & \text{if } p = s_i \in S \wedge s_i \in F_- \\ [\mathrm{bin}_k(i)b] & \text{if } p = g_i \in G \vee p = s_i \in S \setminus (F_+ \cup F_-) \\ [\mathrm{bin}_k(0)b] & \text{if } p = \lambda \vee p = \# \end{cases}$$

2. An ordered subset $F = \{f_1, \ldots, f_m\} \subseteq S$ is encoded by the concatenation of the codes of its elements:

$$code_F(F) = code_S(f_1) \cdots code_S(f_m)$$

3. A row of the transition table is now encoded by writing the codes of the symbols on 7 tracks which implies an alphabet C^7 for the code:

$$\begin{aligned} code_r\ &(s_i, s_j, s_k, g_l, s_m, g_n, g_o) = \\ &\big(proj_1(code_S(s_i)), proj_1(code_S(s_j)), \ldots, proj_1(code_S(g_o))\big) \\ &\quad\vdots \\ &\big(proj_{k+3}(code_S(s_i)), proj_{k+3}(code_S(s_j)), \ldots, proj_{k+3} code_S(g_o))\big) \end{aligned}$$

4. Consequently, the whole table with m rows $r_1, \ldots, r_m$ is encoded as the concatenation of the codes of the rows:

$$code_\delta = code_r(r_1) \cdots code_r(r_m)$$

5. Hence, a PDCA $\mathcal{M}$ is encoded by the codes of the state $S_{|S|}$ and the stack symbol $g_{|G|}$ on two tracks (which imply the number of states and stack symbols) followed by the codes of F_+, of F_- and of the transition table. The whole is enclosed in square brackets on each track:

$$\begin{aligned} code\ &(\mathcal{M}) = \\ &[^7\big(proj_1(code_S(s_{|S|})), proj_1(code_S(g_{|G|}))\big) \\ &\quad\vdots \\ &\big(proj_{k+3}(code_S(s_{|S|})), proj_{k+3}(code_S(g_{|G|}))\big) \\ &code_F(F_+) code_F(F_-) code_\delta(\delta)]^7 \end{aligned}$$

For easier reading we regard the encoding as a word over C^7 (i.e. all the not used registers are filled with λ).

Example 2. $S = \{s_1, s_2, s_3, s_4\}$, $G = \{\bot, g_2\}$, $F_+ = \{s_2\}$, $F_- = \{s_3, s_4\}$ and $\delta_s(s_1, s_1, s_0, g_1) = s_2$ and $\delta_p(s_1, s_1, s_0, g_1) = \bot$ define the only row of the transition table. The encoding is as depicted in Figure 1.

It is not hard but a technical challenge to prove that the language $\{w \in (C^7)^+ \mid \exists \text{PDCA } \mathcal{M} : w = code(\mathcal{M})\}$ of all PDCA encodings is recognizable by some PDCA in real-time.

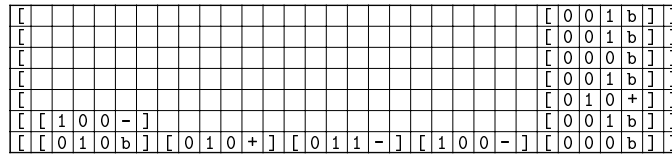


Fig. 1. Example encoding of a PDCA.

4 Encoding of Configurations

By designing an efficiently verifiable encoding of PDCAs we have done the first step towards an efficient universal PDCA. Given the encoding of an arbitrary PDCA $\mathcal{M}$ and the encoding of an input word w a universal PDCA has to simulate the behavior of $\mathcal{M}$ on input w. Since in general w is independent of $\mathcal{M}$ at first it will be necessary to create the encoding of the initial configuration of $\mathcal{M}$ with respect to w. Since the universal PDCA has to handle PDCAs with arbitrarily large state sets the space requirement may be arbitrarily large during the simulation. Since the number of available cells is bounded by the length of the input the encodings have to be stored into the stacks.

The following lemma solves a pattern transformation problem that will be utilized for the generation of initial configurations.

Lemma 3. *[8] Let A be an alphabet, $\mathtt{b} \notin A$, $w = a_0 \cdots a_{n-1} \in A^+$ and $m \in \mathbb{N}$. Then there exists a* PDCA *that transforms a configuration $c_0(i) = a_i$, $0 \leq i \leq n-1$, and $c_0(i) = \mathtt{b}$, $n \leq i \leq n+m$, into the configuration $c_t(i) = a_{i \bmod n}$, $0 \leq i \leq n+m$, within $t \leq 4(n+m)$ time steps.*

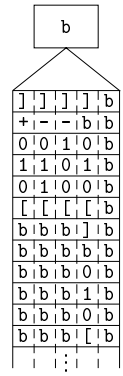


Fig. 2. Example encoding of a cell.

The encoding of a single cell in a configuration is an element from $\{\mathtt{b}\} \times ((C^5)^k)^+$ and, thus, well-suited for later processing. The second component is the stack content, which in turn consists of the codes of the current states of the cell itself and of its neighbors and the current stack content itself.

These codes are written on 5 tracks from top to bottom in reversed form (cf. Example 4). The fifth track is for later use and initially empty. So are the tracks 1, 2 and 3 lower down the stack since the codes for the states are stored only once. Let $k = |code_S(s)|$ be the length of the codes for states. For a cell i at time t let $c_t(i-1) = (s_h, p_{h,1} \cdots p_{h,m_h})$, $c_t(i) = (s_i, p_{i,1} \cdots p_{i,m_i})$ and $c_t(i+1) = (s_j, p_{j,1} \cdots p_{j,m_j})$. Then the encoding of cell i at time t is:

$$
\begin{array}{l}
code_c(i,t) = \\
\mathtt{b}, \\
\big(proj_k(code_S(s_h)), proj_k(code_S(s_i)), proj_k(code_S(s_j)), proj_k(code_S(p_{i,1})), \mathtt{b}\big) \\
\quad \vdots \\
\big(proj_1(code_S(s_h)), proj_1(code_S(s_i)), proj_1(code_S(s_j)), proj_1(code_S(p_{i,1})), \mathtt{b}\big) \\
\big(\mathtt{b}, \mathtt{b}, \mathtt{b}, proj_k(code_S(p_{i,2})), \mathtt{b}\big) \\
\quad \vdots \\
\big(\mathtt{b}, \mathtt{b}, \mathtt{b}, proj_1(code_S(p_{i,2})), \mathtt{b}\big) \\
\big(\mathtt{b}, \mathtt{b}, \mathtt{b}, proj_k(code_S(p_{i,3})), \mathtt{b}\big) \\
\quad \vdots \\
\big(\mathtt{b}, \mathtt{b}, \mathtt{b}, proj_1(code_S(p_{i,m_i})), \mathtt{b}\big)
\end{array}
$$

Consequently, the encoding of a configuration c_t is the concatenation of the encodings of the cells:

$$code_c(c_t) = code_c(1,t)code_c(2,t) \cdots code_c(n,t)$$

Example 4. $S = \{s_1, s_2, s_3, s_4\}$, $G = \{\bot, g_2\}$, $F_+ = \{s_2\}$, $F_- = \{s_3, s_4\}$. Let $c_t(i-1) = (s_2, g_2\bot)$, $c_t(i) = (s_3, g_2g_2\bot)$ and $c_t(i+1) = (s_4, \bot)$ then $code_c(i,t)$ is:

$$
\begin{array}{l}
\mathtt{b}, \\
(\mathtt{]},\mathtt{]},\mathtt{]},\mathtt{]},\mathtt{b})(\mathtt{+},\mathtt{-},\mathtt{-},\mathtt{b},\mathtt{b})(\mathtt{0},\mathtt{0},\mathtt{1},\mathtt{0},\mathtt{b})(\mathtt{1},\mathtt{1},\mathtt{0},\mathtt{1},\mathtt{b})(\mathtt{0},\mathtt{1},\mathtt{0},\mathtt{0},\mathtt{b})(\mathtt{[},\mathtt{[},\mathtt{[},\mathtt{[},\mathtt{b}) \\
(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{]},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{0},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{1},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{0},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{[},\mathtt{b}) \\
(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{]},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{0},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{0},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{0},\mathtt{b})(\mathtt{b},\mathtt{b},\mathtt{b},\mathtt{[},\mathtt{b})
\end{array}
$$

It will be stored in a single cell and its stack as depicted in Figure 2.

5 Universal Pushdown Cellular Automata

This section is devoted to the construction of an efficient universal PDCA. The following first construction yields a PDCA $\mathcal{U}$ that, given the encoding of a PDCA $\mathcal{M}$ and the encoding of a configuration c_t of $\mathcal{M}$, computes the encoding of the successor configuration c_{t+1} of $\mathcal{M}$ within some r time steps.

$\mathcal{U}$ simulates one transition step of $\mathcal{M}$ in three phases. During the *presimulation phase* some of the tracks are initialized. The successor states and stack symbols of $\mathcal{M}$ are computed during the *simulation phase*. Subsequently, during

the *postsimulation phase* a valid configuration is computed such that the next step of $\mathcal{M}$ can be simulated.

Presimulation phase: $\mathcal{U}$ has five main tracks which may be divided into subtracks. If in the sequel the lengths of some blocks do not divide the number of cells or if the construction uses a constant number of cells beyond the borders it is assumed that there exist another five main tracks that are used as the folded extensions of the corresponding tracks in order to cope with that (constant) space problem.

The cell that contains the last symbol of $code(\mathcal{M})$ can identify itself by the situation $\texttt{]}^7\texttt{]}^7$. The same holds for the cell that contains the first symbol of $code_\delta(\delta)$ by the situation $\texttt{][}^7$ (cf. Figure 1).

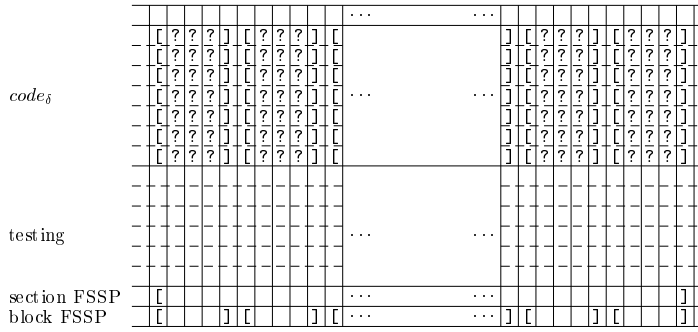


Fig. 3. An inner section after the presimulation phase.

The cells that contain $code_c(c_t)$ are divided into sections of length $|code_\delta(\delta)|$ such that $code_\delta(\delta)$ can be made available on track 2 in each section. Each section is divided into blocks of length $|code_S(s)|$, lets say of length k. The result of the presimulation phase is depicted in Figure 3.

By Lemma 3 the sections can be created by concatenations of $code_\delta(\delta)$. During that process the inscriptions of track 4 and 5 can also be generated.

At the end of the presimulation phase a global FSSP is started on the first track such that all the cells synchronously start the simulation phase. This takes $2(|code(\mathcal{M})| + |code_c(c_t)|)$ time steps. Thus, altogether the presimulation phase needs $t \leq 4(|code(\mathcal{M})|+|code_c(c_t)|)+2(|code(\mathcal{M})|+|code_c(c_t)|) = 6(|code(\mathcal{M})|+|code_c(c_t)|)$ time steps.

Simulation phase: In the sequel all signals and labellings are realized on the first main track (resp. its subtracks). The simulation is performed in all sections in parallel. Therefore, it suffices to explain the process for one section.

On the fifth track modified FSSPs are performed. `[` and `]` are generals that synchronize the cells in between them every k (i.e. $code_S(s)$) time steps. After each other synchronization the process is one time step delayed.

During such a cycle all cells of $code_c(c_t)$ with $\texttt{]}^7$ on their second track are working as follows (cf. Figure 4 for the general behavior).

During each time step the top-of-stack symbol is copied onto five subtracks of the third track. At the same time the contents of the tracks 2 and 3 are shifted

to the right. Since on the second track there is the encoding of one of the rows of the transition table the cell can successively test whether the row matches its current situation.

After k time steps the FSSP fires and the contents of the second and third track now are successively shifted to the left. Therefore, the cell can push its old state and top-of-stack symbol back into its stack if the row did not match the current situation, or the new state and top-of-stack symbol(s) otherwise. The test is finished when the FSSP fires again. During the delay step the contents of track 2 and 5 are shifted one cell to the right and the next cells will be tested during the next $2k$ time steps.

The process has to be repeated until all cells of the section have been tested with all blocks of $code_\delta(\delta)$. This needs $|code_\delta(\delta)| \cdot (2k + 1)$ time steps.

In order to recognize the end of the simulation phase on track 4 a FSSP is performed that synchronizes the whole section. One transition of the FSSP is computed at every time the block FSSP has finished a test cycle. Thus, the section is exactly synchronized after $|code_\delta(\delta)| \cdot (2k + 1)$ time steps.

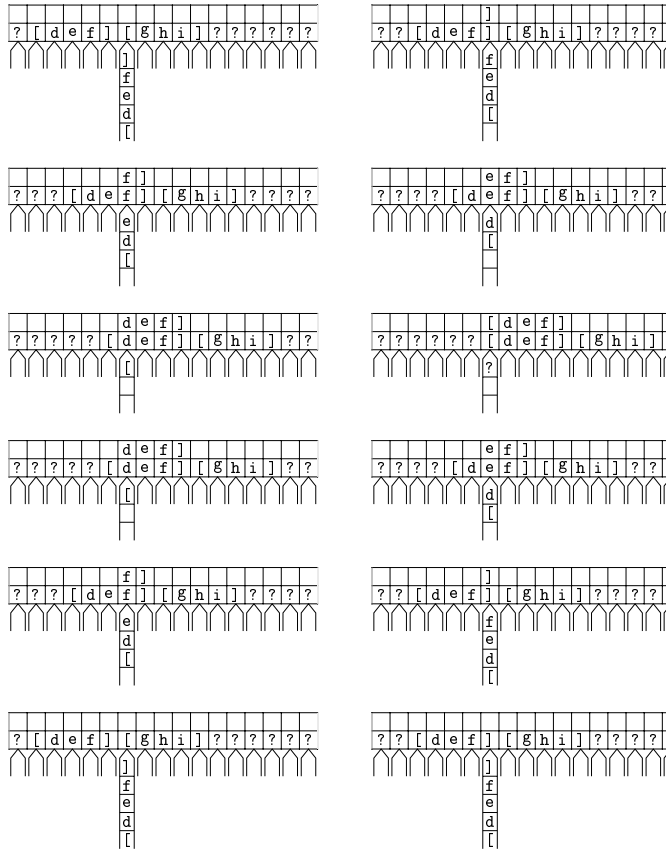


Fig. 4. A test cycle in the simulation phase.

Postsimulation phase: At the beginning of this phase the situation at the top of the stacks is as depicted in Figure 5.

The process that updates the stack contents is similar to the simulation phase. During k time steps the contents of the top of the stack of three adjacent cells are successively copied onto the third track. Then during another $2k$ time steps the stack content of the inner cell is updated appropriately.

The postsimulation phase needs k such update cycles, hence, $k(3k+1)$ time steps.

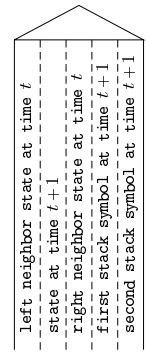


Fig. 5. Stack content at the beginning of the postsimulation phase.

Theorem 5. *Let n denote the length of the input of $\mathcal{U}$, then $\mathcal{U}$ simulates l transitions of a* PDCA $\mathcal{M}$ *within $t \leq 6n + 7l|code_\delta(\delta)|^2$ time steps.*

Proof. The theorem follows from the fact that the presimulation phase has only to be performed once. Let $k = |code_S(s)|$. From the construction we obtain:

$$\begin{aligned} t &\leq 6(|code(\mathcal{M})| + |code_c(c_t)|) + l(|code_\delta(\delta)| \cdot (2k+1) + k(3k+1)) \\ &\leq 6n + l(|code_\delta(\delta)| \cdot 2k + |code_\delta(\delta)| + 3k^2 + k) \\ &\leq 6n + l(2|code_\delta(\delta)|^2 + |code_\delta(\delta)| + 3|code_\delta(\delta)|^2 + |code_\delta(\delta)|) \\ &\leq 6n + l(5|code_\delta(\delta)|^2 + 2|code_\delta(\delta)|) \\ &\leq 6n + 7l|code_\delta(\delta)|^2 \end{aligned}$$

□

The PDCA $\mathcal{U}$ works fine if its input is the encoding of a PDCA $\mathcal{M}$ and the encoding of a configuration of $\mathcal{M}$. In the following this precondition is too restrictive since for diagonalization proofs we need to consider the behavior of PDCAs when they get their own encoding as input.

The following construction yields a PDCA $\mathcal{V}$ that, given the encoding of a PDCA $\mathcal{M}$, computes the encoding $code_c(c_0)$ where c_0 is the initial configuration of $\mathcal{M}$ with input $code(\mathcal{M})$. Subsequently, $\mathcal{V}$ computes the encoding of the successor configuration c_1 of $\mathcal{M}$ and so on.

The first task of $\mathcal{V}$ is to compute the encoding of the initial configuration of $\mathcal{M}$. The second task is to simulate the universal PDCA $\mathcal{U}$ in order to compute encodings of successor configurations of $\mathcal{M}$.

The construction of $\mathcal{U}$ has been done under the assumption that $code(\mathcal{M})$ is located at the left of the encoding of the configuration. This situation can be emulated as follows. We assume that for each track there exists another one which is regarded as the extension of the track. Both tracks are connected at the left border. So it suffices to write the mirror image of $code(\mathcal{M})$ on the extension. This can be done in $|code(\mathcal{M})|$ time steps.

Subsequently, $\mathcal{V}$ simulates the presimulation phase of $\mathcal{U}$ where $code_\delta(\delta)$ is concatenated in order to initialize the sections. Parallel to this process all cells i compute $code_c(i, 0)$ which completes the first task of $\mathcal{V}$. This computation is now explained for one of the stack registers.

During the presimulation phase $code_\delta(\delta)$ is moved across the cells. Cell i with input $s_i \in C$ "knows" $\mathrm{bin}(i)$. When it receives a `]` it waits for $|\mathrm{bin}(i)|$ time steps and subsequently pushes `0`s into the stack until it receives a `[`. Now it pushes $\mathrm{bin}(i)$ into the stack.

If $code_F(F_+)$ and $code_F(F_-)$ are also moved across the cells, the encoding of s_i can be completed simply by successively testing whether s_i belongs to one of the sets and by pushing the appropriate symbol `+`, `-` or `b` followed by a `[`.

Theorem 6. *$\mathcal{V}$ simulates l transitions of a* PDCA *$\mathcal{M}$ that operates on its own encoding within $t \leq 13n + 7l|code_\delta(\delta)|^2$ time steps, where n denotes the length of the input of $\mathcal{V}$.*

Proof. From the construction follows: $\mathcal{U}$ needs $|code(\mathcal{M})| = n$ time steps to create the mirror image of $code(\mathcal{M})$. Subsequently, $\mathcal{V}$ simulates the presimulation phase of $\mathcal{U}$ in which additionally the encoding of the configuration is computed. Since we are concerned with extended tracks the time has to be doubled. Thus, this phase needs $12n$ time steps. After the presimulation phase $\mathcal{V}$ simulates $\mathcal{U}$ directly and the theorem follows. □

6 Tight Hierarchies of Language Families

Subject of this section are tight time and space hierarchies. Since the number of cells is fixed by the length of the input the space complexity is measured as stack-space.

Definition 7. *Let $\mathcal{M} = \langle S, G, \delta_s, \delta_p, \#, \bot, A, F_+, F_- \rangle$ be a* PDCA. *$F_+ \cup F_-$ is the set of final states.*

1. *A word $w \in A^+$ is* accepted *resp.* rejected *by $\mathcal{M}$ if at input w the leftmost cell of $\mathcal{M}$ becomes final and if its first final state is an accepting resp. rejecting state.*
2. *$L(\mathcal{M}) = \{w \in A^+ \mid w$ is accepted by $\mathcal{M}\}$ is the* language accepted *by $\mathcal{M}$.*
3. *Let $t : \mathbb{N} \to \mathbb{N}$, $t(n) \geq n$, be a function. A* PDCA *is said to be* t-time-bounded *or of* time complexity t *iff every input of length n after at most $t(n)$ time steps is accepted or rejected.*

4. *Let $g : \mathbb{N} \to \mathbb{N}$ be a function. A* PDCA *is said to be* g-stack-space-bounded *or of* space complexity g *iff every input of length n is accepted or rejected at some time t and for all $t' \leq t$ each of the stacks contains at most $g(n)$ symbols.*

The family of all languages which can be accepted by PDCAs with time complexity t resp. space complexity g is denoted by $\mathscr{L}_t(\text{PDCA})$ resp. ${}_g\mathscr{L}(\text{PDCA})$.

In order to prove infinite tight hierarchies in almost all cases honest resource bounding functions are required. Usually the notion "honest" is concretized in terms of the computability or constructibility of the function with respect to the device in question.

Definition 8. *A function $f : \mathbb{N} \to \mathbb{N}$ is* computable *if there exists a* PDCA $\mathcal{M}$ *with input alphabet A and constant stack-space complexity such that $\mathcal{M}$ recognizes all $w \in A^+$ in exactly $f(|w|)$ time steps.*

Thus, computability of f means that there exists a PDCA that for any input w from A^+ can distinguish the time step $f(|w|)$ without using its stacks (i.e. a classical cellular automaton). As usual here we remark that the class of such functions is very rich [2,3,6,13].

Now we are prepared to prove the tight time hierarchy.

Theorem 9. *Let $t : \mathbb{N} \to \mathbb{N}$ and $t' : \mathbb{N} \to \mathbb{N}$ be two functions. If t is computable and $\liminf_{n\to\infty} \frac{t'(n)}{t(n)} = 0$ then there exists a language L such that*

$$L \in \mathscr{L}_{t(n)}(\text{PDCA}) \setminus \mathscr{L}_{t'(n)}(\text{PDCA})$$

If additionally $\forall n \in \mathbb{N} : t'(n) \leq t(n)$ then $\mathscr{L}_{t'(n)}(\text{PDCA}) \subset \mathscr{L}_{t(n)}(\text{PDCA})$.

Proof. Let $\mathcal{W}$ be a PDCA that works as follows. At first $\mathcal{W}$ checks whether or not its input belongs to the language $L' = \{uv \mid v \in \{0,1\}^+ \wedge \exists \text{PDCA } \mathcal{M} : u = code(\mathcal{M})\}$. Since the cell that contains the last symbol of u can identify itself this verification needs at most $|uv|$ time steps.

Subsequently, $\mathcal{W}$ performs two tasks in parallel. One is to simulate the computation of $\mathcal{M}$ with input uv as has been shown by the construction for Theorems 5 and 6. The second one is to distinguish the time step $t(|uv|)$ since t is computable.

$\mathcal{W}$ rejects its input if $uv \notin L'$ or if after $t(|uv|)$ time steps the simulation of $\mathcal{M}$ has not produced a decision. If, on the other hand, $\mathcal{W}$ recognizes during $t(|uv|)$ time steps that $\mathcal{M}$ did its decision, then $\mathcal{W}$ rejects if $\mathcal{M}$ accepts and vice versa. Thus, $L(\mathcal{W}) \in \mathscr{L}_t(\text{PDCA})$.

Contrarily to the assertion we assume that there exists a PDCA $\mathcal{W}'$ with $u = code(\mathcal{W}')$ that recognizes $L(\mathcal{W})$ with time complexity t'. By Theorem 6 $\mathcal{W}$ needs $k_1|uv| + k_2 t'(|uv|)k_3^2$ time steps in order to simulate $t'(|uv|)$ transitions of $\mathcal{W}'$. k_1, k_2 and k_3 are constants that depend on $\mathcal{W}'$. W.l.o.g. we may assume $t'(|uv|) \geq |uv|$. Therefore, the time complexity of $\mathcal{W}$ is $k_1|uv| + k_2 t'(|uv|)k_3^2 \leq k_4 t'(|uv|)$ for a suitable constant k_4. From the limes inferior we obtain a $v' \in \{0,1\}^+$

such that $k_4 t'(|uv'|) \leq t(|uv'|)$. Therefore, $\mathcal{W}$ can simulate $t'(|uv'|)$ transitions of $\mathcal{W}'$ within $t(|uv'|)$ time steps. If $\mathcal{W}'$ accepts the input uv' in $t'(|uv'|)$ time steps then $\mathcal{W}$ rejects. If $\mathcal{W}'$ rejects the input within $t'(|uv'|)$ time steps then $\mathcal{W}$ accepts. This is a contradiction to the assumption that $\mathcal{W}'$ accepts $L(\mathcal{W})$ with time complexity t'. □

Without proof we state the tight space hierarchy:

Theorem 10. *Let $g : \mathbb{N} \to \mathbb{N}$ and $g' : \mathbb{N} \to \mathbb{N}$ be two functions. If g is computable and $\liminf_{n\to\infty} \frac{g'(n)}{g(n)} = 0$ then there exists a language L such that*

$$L \in {}_{g(n)}\mathscr{L}(\text{PDCA}) \setminus {}_{g'(n)}\mathscr{L}(\text{PDCA})$$

If additionally $\forall n \in \mathbb{N} : g'(n) \leq g(n)$ then ${}_{g'(n)}\mathscr{L}(\text{PDCA}) \subset {}_{g(n)}\mathscr{L}(\text{PDCA})$.

References

1. Albert and Čulik II, K. *A simple universal cellular automaton and its one-way and totalistic version*. Complex Systems 1 (1987), 1–16.
2. Buchholz, Th. and Kutrib, M. *Some relations between massively parallel arrays*. Parallel Comput. 23 (1997), 1643–1662.
3. Buchholz, Th. and Kutrib, M. *On time computability of functions in one-way cellular automata*. Acta Inf. 35 (1998), 329–352.
4. Harrison, M. A. *Introduction to Formal Language Theory*. Addison-Wesley, Reading, 1978.
5. Imai, K. and Morita, K. *A computation-universal two-dimensional 8-state triangular reversible cellular automaton*. Theoret. Comput. Sci. 231 (2000), 181–191.
6. Iwamoto, C., Hatsuyama, T., Morita, K., and Imai, K. *On time-constructible functions in one-dimensional cellular automata*. Fundamentals of Computation Theory 1999, LNCS 1684, 1999, pp. 317–326.
7. Kutrib, M. *Pushdown cellular automata*. Theoret. Comput. Sci. 215 (1999), 239–261.
8. Kutrib, M. *Efficient Universal Pushdown Cellular Automata and their Application to Complexity*. IFIG Research Report 0004, Institute of Informatics, University of Giessen, Giessen, 2000.
9. Kutrib, M. and Richstein, J. *Real-time one-way pushdown cellular automata languages*. Developments in Language Theory II. At the Crossroads of Mathematics, Computer Science and Biology, World Scientific, Singapore, 1996, pp. 420–429.
10. Margenstern, M. *Frontier between decidability and undecidability: a survey*. Theoret. Comput. Sci. 231 (2000), 217–251.
11. Martin, B. *Efficient unidimensional universal cellular automaton*. Mathematical Foundations of Computer Science 1992, LNCS 629, 1992, pp. 374–382.
12. Martin, B. *A universal cellular automaton in quasi-linear time and its S–m–n form*. Theoret. Comput. Sci. 123 (1994), 199–237.
13. Mazoyer, J. and Terrier, V. *Signals in one dimensional cellular automata*. Theoret. Comput. Sci. 217 (1999), 53–80.
14. Morita, K. *Computation-universality of one-dimensional one-way reversible cellular automata*. Inform. Process. Lett. 42 (1992), 325–329.
15. Morita, K. and Ueno, S. *Computation-universal models of two-dimensional 16-state reversible cellular automata*. Trans. IEICE 75 (1992), 141.

Firing Squad Synchronization Problem on Bidimensional Cellular Automata with Communication Constraints

Salvatore La Torre[1, 2], Margherita Napoli[2], and Mimmo Parente[2]

[1] University of Pennsylvania, USA
[2] Università degli Studi di Salerno, Italia
parente@unisa.it

Abstract. We are given a network of identical processors that work synchronously at discrete time steps, a Cellular Automaton. Processors are arranged as an array of m rows and n columns with the constraint that they can exchange only one bit of information with their neighbours. We study the problem to synchronize the cellular automata, that is to let all the processors enter the same state for the first time at the very same instant. This problem is also known as "The Firing Squad Synchronization Problem", introduced by Moore in 1964. We give algorithms to synchronize two dimensional Cellular Automata of $(n \times n)$ cells at some given times: n^2, $n\lceil \log n \rceil$, $n\lceil \sqrt{n}\, \rceil$ and 2^n. Moreover we also give some general constructions to synchronize Cellular Automata of $(m \times n)$ cells. Our approach is a modular description of synchronizing algorithms in terms of "fragments" of cellular automata that are called signals.

1 Introduction

The *Firing Squad Synchronization Problem* (shortly FSSP) is the problem of synchronizing a Cellular Automaton (CA), a network of identical cells (finite automata) that work synchronously at discrete time steps. A general formulation of this problem is the following: initially a distinguished cell (the *general*) starts computing while all others are in a quiescent state; at each time step any cell sends/receives to/from its neighbours some information about their state at the preceding time; the problem is to let all cells in the network enter the same state, called *firing*, for the first time at the very same instant.

The FSSP was introduced by Moore in 1964 [6] as the problem of synchronizing a linear CA, where at each step each cell transmits its current state to its two adjacent cells. (Its name is due to the fact that the line of cells can be seen as a line of soldiers that have to fire simultaneously.) The early results all focused on the synchronization in minimal time of a linear CA. A significant amount of papers also dealt with some variations of the FSSP that concerned both the geometry of the network and computational constraints. Besides minimal-time solutions to the FSSP, also solutions at a predetermined (non minimal) time have been considered [3,4]. This is an interesting and challenging theoretical

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 264–275, 2001.

problem, which is also directly connected to the sequential composition of cellular automata. Given two cellular automata A_1 and A_2 computing respectively the functions $f_1(x)$ and $f_2(x)$, the sequential composition of A_1 followed by A_2 is the cellular automaton obtained in the following way: first A_1 starts on a standard initial configuration and when it has done with its computation, A_2 starts using the final configuration of A_1 as initial configuration. The resulting automaton clearly computes $f_2(f_1(x))$. In order to compose the two automata, it is necessary to synchronize all the cells that will be used by A_2 at the time A_1 computes $f_1(x)$ for a given input x. Solutions to the FSSP at a given time have been studied on rings and toroidal square arrays with unidirectional flows of information [3], and on lines of cells exchanging only one bit at each time step [4].

In this paper we study non minimal time solutions to the FSSP. We consider a one bit 2-dimensional cellular automaton, that is a grid of $(m \times n)$ identical finite-state processors (*cells*) which exchange one bit of information each other. A *synchronization in time* $t(m,n)$ of an array of $(m \times n)$ cells is a one bit 2-dimensional cellular automaton such that, starting from a standard configuration, all cells enter for the first time the Firing state at time $t(m,n)$. We obtain new synchronization times in a compositional way: we first describe basic synchronizing algorithms and then we give general rules to compose synchronizations. The basic synchronizations in turn are obtained by composing elementary signals, which can be seen as fragments of Cellular Automata. A *synchronization* is thus a special signal obtained as a composition of many simpler signals. Compositional rules for both signals and synchronizations include *parallel* composition, *sequential* composition, and *iterated* composition. We also state some sufficient conditions to apply these compositions. In the parallel composition we start many synchronizations or signals, all at the same time. Sequential composition appends a synchronization or a signal to the end of another signal, possibly with a constant time offset. This way we are able to construct a synchronization in time $t_1(m,n) + t_2(m,n) + d$, for $d \geq 0$, if there exist synchronizations in time $t_1(m,n)$ and $t_2(m,n)$. The iterated composition consists of iterating $t_2(m,n)$ times a synchronization in time $t_1(m,n)$ given a synchronization in time $t_2(m,n)$, thus obtaining a new synchronization in time $t_1(m,n) \cdot t_2(m,n)$.

We also give a construction to "inherit" synchronizations on $(m \times n)$ arrays of processors from synchronizations of lines of k processors. We show that an $(m \times n)$ array of processors can be seen as many lines of $(m+n-1)$ processors (each of them having as endpoints cells $(1,1)$ and (m,n)) where a synchronization can be executed simultaneously on all these lines. Thus we can synchronize an $(m \times n)$ array in time $t(m+n-1)$, provided that there exists an algorithm for a linear array of k processors in time $t(k)$. Finally, we give algorithms to synchronize an $(n \times n)$ square array in time n^2, $n\lceil \log n \rceil$, $n\lceil \sqrt{n} \rceil$ and 2^n, and we compositions of synchronizations to determine synchronizations in any "feasible" linear time and in any time expressed by a polynomial with nonnegative coefficients. These constructions use as building block synchronizations in time n^2 and in minimal time.

2 Preliminaries

In this section we give the definitions needed in the rest of the paper.

Basic Definitions. A one bit 1-dimensional cellular automaton (shortly 1-CA) is a line of n finite-state machines, called *cells*, which are identical except for those at the two endpoints. In a 1-CA, the i-th cell is connected to the $(i-1)$-th and $(i+1)$-th cells, for all $i = 2, \ldots, n-1$. The first and the last cells are connected, respectively, to the second and the $(n-1)$-th cell. The 2-dimensional case is a natural generalization of the 1-CA. Omitting minor details, a one bit 2-dimensional cellular automaton (shortly 2-CA) is an array of $(m \times n)$ finite-state machines (*cells*) which are identical except for the boundary ones, and where cell (i, j) is connected to cells $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. The cells operate synchronously at discrete time steps. At each step each cell exchanges one bit of information with its adjacent cells and modifies its state depending on its current state and the bits sent by the adjacent cells at the previous step.

In what follows, the symbol Q refers to the set of states of a given cellular automaton. A *configuration* of a 1-CA is a mapping $C : \{1, 2, \ldots, n\} \to \{0, 1\} \times Q \times \{0, 1\}$. A configuration at time t gives, for each cell i, the state entered and the two bits sent at this time. A starting configuration is a configuration at time 1. The definition of configuration can be easily extended to a 2-CA, by considering that each cell sends the bits to its four adjacent cells. In the following we often write "(A, C)" to denote a 1-CA, or a 2-CA, A starting on a configuration C. Within the state set there are three distinguished state: G the *General* state, L the *Latent* state, and F the *Firing* state. State L has the property that if a cell in state L receives all bits 0 from its neighbours, it remains in the state L and sends bits 0 to its neighbours. A *standard configuration* is a configuration where each cell is in state L and sends bits 0, except for cell 1 (resp. cell $(1, 1)$) which is in state G and sends bits 1 to each neighbour. A *synchronization in time* $t(n)$ of a linear array of n cells, is a 1-CA such that starting from a standard configuration all cells enter at time $t(n)$ for the first time state F. Analogously a *synchronization in time* $t(m, n)$ of a rectangular array of $(m \times n)$ cells is a 2-CA such that starting from a standard configuration all cells enter for the first time state F at time $t(m, n)$. When $m = n$, we speak about a synchronization of a square array in time $t(n)$. A synchronization of a linear array of n cells in time $(2n-1)$ is called a minimal time synchronization, since it can be easily proved that a synchronization is not possible in time less than $(2n-1)$.

Signals. In [4] the concept of signal was introduced as a mean to design a 1-CA. Informally a signal describes the information flow in the space-time description of a cellular automaton, allowing a modular description of the synchronization process. The scheme used to present some synchronization algorithms in time $t > 2n-1$ for a a linear array of n processors is the following: some signals are designed and composed in order to obtain an overall signal that starts from the leftmost processor and comes back to it in exactly $(t-2n+1)$ time units; then a minimal time synchronization starts, thus synchronizing the n processors in time t. We consider the time unrolling of a 1-CA A starting on a configuration

C, that is we reason about a space-time array. A pair (i,t) of this array, with $1 \leq i \leq n$ and $t \geq 1$, is called a *site*, the state of the cell i at time t is denoted *state*(i,t) and the bits sent to the adjacent cells are denoted by *left*(i,t) and *right*(i,t). A site (i,t) is said to be *active* if either sends/receives a bit 1 to/from its neighbours or changes its state. We denote by Cell(A,C) the set of cells i such that site (i,t) is active for some t.

Let A be a 1-CA and C be a configuration. Define the time $t_i^{\max} = \max\{t|(i,t)$ is active$\}$ and $t_i^{\min} = \min\{t|(i,t)$ is active$\}$. The set of sites $(i,t_i^{\min})$ for $i \in$ Cell(A,C) is called *rear* of (A,C) and the set of sites $(i,t_i^{\max})$ is the *front* of (A,C). Moreover we say that (A,C) is *tailed* if there exists a subset of Q, called *tail*(A,C) such that for all $i \in \{1,\ldots,n\}$, *state*$(i,t) \in$ tail(A,C) if and only if (i,t) belongs to the front of (A,C). The states in tail(A,C) are called *tail* states. In words, a tail state appears for the first time on the front of (A,C).

Two active sites $(i_1,t_1),(i_2,t_2)$ are *consecutive* if $t_2 = t_1 + 1$ and $i_2 \in \{i_1 - 1, i_1, i_1 + 1\}$. A *simple signal* of (A,C) is a subset S of consecutive sites with the property that if (A,C) is tailed, then $(i,t_i^{\max})$ belongs to S. The union of a finite number of simple signals of a given (A,C) is called *signal* of (A,C). A graphical representation of a simple signal S is obtained by drawing a line between:
(i) every pair of sites $(i,t) \in S$ and $(i,t+1) \in S$ and
(ii) every pair of sites $(i,t) \in S$ and $(i+1,t+1) \in S$ (resp. $(i-1,t+1) \in S$) if *right*$(i,t) = 1$ (resp. *left*$(i,t) = 1$).
A graphical representation of a signal is obtained by the graphical representation of its simple signals. The *length* of a signal S is $(t^{\max} - t^{\min} + 1)$ where $t^{\max} = \max\{t|(i,t) \in S, 1 \leq i \leq n\}$ and $t^{\min} = \min\{t|(i,t) \in S, 1 \leq i \leq n\}$. Sometimes, in the rest of paper we refer to a signal without specifying a 1-CA and a starting configuration.

In the following example we recall a signal introduced in [4].

Example 1. Given a positive constant $k < n$, the signal MARK$(n-k)$ is used to mark the cell $n-k$. The length of the signal MARK is $n+k$(see Figure 1). It can be easily seen that MARK is a signal of a tailed 1-CA.

We recall now the signal composition. We say that a 1-CA A_2 on C_2 *can follow* a tailed 1-CA A_1 on C_1 if there exists a function h defined over tail(A_1,C_1) and such that $h(p) = C_2(i)$ if $p =$ *state*(i,t). Given two signals S_1 and S_2, we can define the concatenation $\text{cat}_r(S_1,S_2)$ as the signal obtained by starting S_1 at time 1 and S_2 at time $r+1$, that is S_2 is delayed r time steps. More formally $\text{cat}_r(S_1,S_2) = S_1 \cup \{(i,t+r)|(i,t) \in S_2\}$.

The following remark recalls some sufficient conditions for the existence of a tailed 1-CA for a signal $\text{cat}_r(S_1,S_2)$.

Remark 1. [4] Let S_1,S_2 be signals of tailed 1-CA's (A_1,C_1) and (A_2,C_2), respectively. The signal $S = \text{cat}_r(S_1,S_2)$ is the signal of a tailed 1-CA (A,C) if the following two conditions hold:

1. (A_2,C_2) can follow (A_1,C_1);
2. if the site (i,t) belongs to the front of (A_1,C_1) and (i,t') belongs to the rear of (A_2,C_2), then $t < t' + r$.

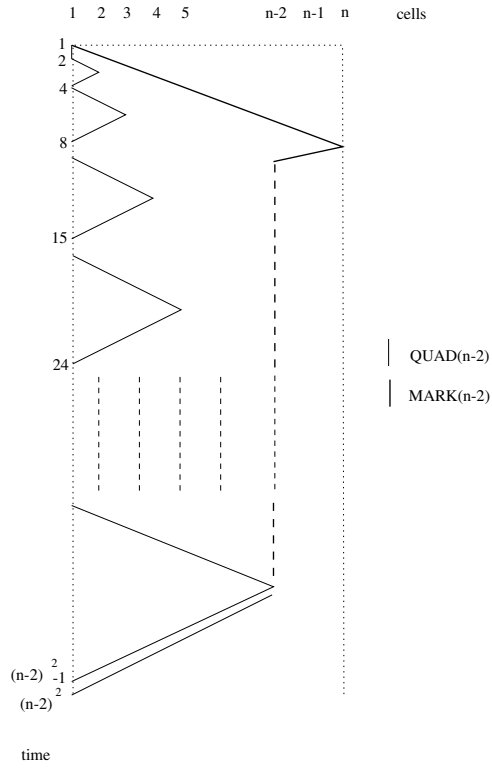


Fig. 1. The signal $\mathrm{cat}_1(\textsc{Quad}(n-2), \textsc{Mark}(n-2))$.

3 Synchronization of a Square Array

In this section we introduce two signals of a 1-CA. The first has a quadratic length and the second has an exponential length in the number of cells. Then we design synchronizations of a square array using the following scheme: first we synchronize the first row of the square array and then we simultaneously synchronize all the columns. That is the rows and the columns are seen as 1-CA. This way we obtain synchronizations of a square array of $(n \times n)$ cells in n^2, 2^n, $n\lceil \log n \rceil$ and $n\lceil \sqrt{n}\, \rceil$.

The signal Quad**.** Given a positive constant $k < n$, $\textsc{Quad}(n-k)$ is a signal of a 1-CA A which is described as follows:

- initially the cell 1 sends a bit 1 to the right; then if it receives a bit 1 from the right, it sends with a delay of one step (except for the first time, when there is no waiting), a bit 1 back to the right; the cell 1 eventually halts when it receives two consecutive bits 1;
- for $1 < h < (n-k)$, the cell h sends a bit 1 to the left when it receives for the first time a bit 1 from the left; then, if the cell h receives again a bit 1 from an adjacent cell, it sends a bit 1 to the other adjacent cell;

– the cell $(n-k)$ sends two consecutive bits 1 to the left when it receives a bit 1 from the left.

The 1-CA A can be further designed such that it is tailed by observing that the cells from 1 to $(n-k)$ can enter a tail state when they receive two consecutive bits 1. The length of the QUAD signal is $(n-k)^2-1$.

Clearly, for the implementation of this signal cell $(n-k)$ needs to be distinguished. In what follows we will use QUAD$(n-2)$, thus we only need to distinguish cell $(n-2)$: this can be done by MARK$(n-2)$ and for all $n > 5$. Clearly for smaller n much easier ad hoc algorithms can be given, (see Figure 1).

The signal EXP. Given two positive constants k and c, we will define the signal EXP$(n-k, c)$.

An *idle* cell is a cell which never sends a bit 1 unless it receives a bit 1 from the left and in this case it sends two consecutive bits 1 to the left.

Initially the only idle cell is the cell $(n-k)$. EXP$(n-k, c)$ is a signal of a 1-CA which is described as follows:

– first cell 1 sends a bit 1 to the right; then, whenever cell 1 receives a bit 1 from the right, it immediately replies sending back a bit 1; finally, if cell 1 receives two consecutive bits 1 from the right, then it changes into an idle cell;
– for $1 < h < (n-k)$, we distinguish two cases:
  • if the bit is received from the left then it alternates the following two behaviours:
    1. it sends a bit 1 back to the left, (let us call these *peak cells*)[1]
    2. it sends a bit 1 to the right;

    each peak cell starts counting from 1 to $2^{i+1}-2$, for $1 < i \leq c$. When $2^{i+1}-2$ has been just counted, if the peak cell receives a bit 1 from the left at the next time unit, then it is the i-th cell in the line and is marked (see below for an explanation). This way it can be distinguished later.
  • if a bit 1 is received from the right, then it sends a bit 1 to the left. If at the next time unit cell h receives another bit 1 from its right neighbour, then two other subcases need to be considered:

    if $h > c$ then the cell switches into an idle cell;
    else, for $h \leq c$, the cell sends two consecutive bits 1 to the left. (Note that when this case occurs, cells $h \leq c$ have already been marked by step 2 above.)

From the algorithm we have just described, a proof by induction on $i \leq c$ can be given to show how a peak cell can be marked, in fact the following property holds: the length of the interval from the instant cell i is a peak cell for the first time and the instant it becomes a peak cell for the second time is $2^i + \sum_{j=1}^{i-1} 2^j(i-j)$, (see Figure 2 where $c = 3$, cell 2 is marked at time 9 and cell 3 is marked at time 20.)

[1] Actually, this is a property of the state entered by this cell.

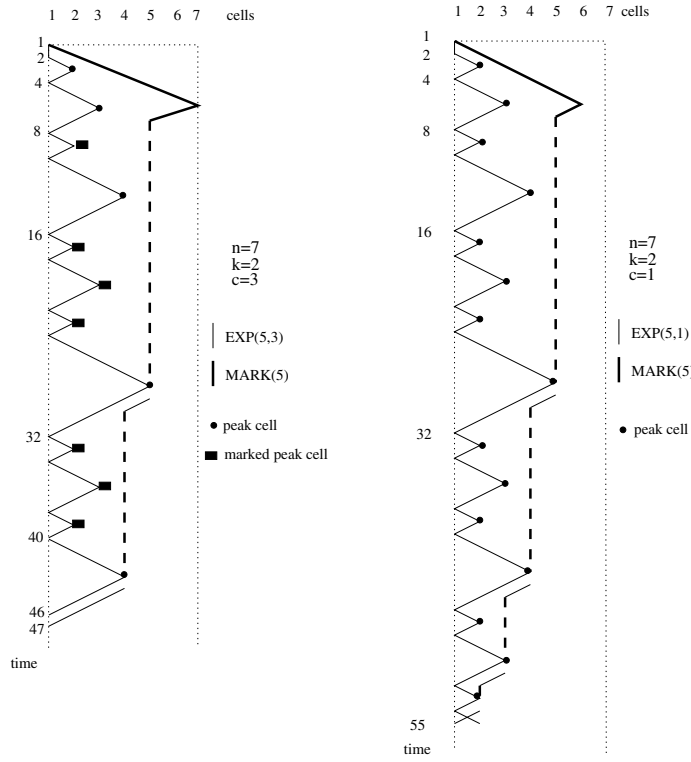


Fig. 2. The signals $\mathrm{cat}_1(\textsc{Exp}(5,3), \textsc{Mark}(5))$ and $\mathrm{cat}_1(\textsc{Exp}(5,1), \textsc{Mark}(5))$

To implement a tailed 1-CA for $\textsc{Exp}(n-k,c)$ initially the cell $(n-k)$ must be distinguished. In what follows we will use the signals $\textsc{Exp}(n-2,3)$ and $\textsc{Exp}(n-2,1)$: the cell $n-2$ can be distinguished by using $\textsc{Mark}(n-2)$, for $n > 5$. Observe also that the cells from 1 to $(n-2)$ can enter a tail state after they received two consecutive bits 1. The length of $\textsc{Exp}(n-k,c)$ is $2^{n-k+1}-2(n-k)-2^{c+1}+2(c+1)$ (see Figure 2).

We can now give the synchronizing algorithms for the square array.

Theorem 1. *There is a synchronization of an $(n \times n)$ square array in time n^2.*

Proof. The algorithm is the following: first a signal $\mathrm{cat}_1(\textsc{Mark}(n-2), \textsc{Quad}(n-2))$ is started on the first row, the length of this signal is $(n-2)^2$ since $\textsc{Quad}(n-2)$ is delayed one time step. This is a signal of a tailed 1-CA starting from a standard configuration (see Remark 1). Thus after $(n-2)^2$ time units the cell $(1,1)$ enters a tail state, say G'. Considering G' as the General state, a minimal time synchronization on a linear array of n cells is executed on the first row and this takes other $(2n-2)$ time units. Once the Firing state F' is reached, we use F' as the General state of a minimal time synchronization that this time runs

on each column, thus taking another $(2n-2)$ time units, which adds up to a total time of n^2.

Theorem 2. *There is a synchronization of an $(n \times n)$ square array in time 2^n.*

Proof. First a signal $\mathrm{cat}_1(\textsc{Exp}(n-2,3), \textsc{Mark}(n-2))$ is started on the first row, see Figure 2. After $(2^{n-1}-2n-3)$ time units the cell $(1,1)$ enters a tail state, say H. This is a signal of a tailed 1-CA starting from a standard configuration (see Remark 1). Now the cell $(1,1)$ enters a state G' and a minimal time synchronization on the first row is accomplished, using G' as the General state, thus taking other $(2n-1)$ time units. Once the Firing state F' is reached, each cell of the first row enters a state G'', and launches the signals $\textsc{Mark}(n-2)$ and $\textsc{Exp}(n-2,1)$ on each column, using G'' as the General state. This takes another $(2^{n-1}-2n+5)$ time units, which sums up to time (2^n-2n+1). Finally, a minimal time synchronization on each column is accomplished, thus reaching time 2^n.

We can construct a synchronization in time $n\lceil \log n \rceil$ and in time $n\lceil \sqrt{n}\, \rceil$ using signals of exponential and of quadratic length, by exploiting the inverse functions. (The algorithms resemble those used to synchronize a line of n cells at the same times shown in [4].)

Theorem 3. *There is a synchronization of a $(n \times n)$ square array in time $n\lceil \log n \rceil$ and in time $n\lceil \sqrt{n}\, \rceil$.*

4 How to Obtain New Synchronizations

In this section we discuss how to obtain new synchronizations of $(m \times n)$ arrays using known algorithms to synchronize linear arrays. We start describing synchronizations of an $(m \times n)$ array in time $t(m+n-1)$, given a synchronization of a line of k processors in time $t(k)$. Then, we give some compositional rules on synchronizations of $(m \times n)$ arrays. We conclude this section showing how to construct synchronizations of a square array of processors in any arbitrary "feasible" linear time and in any time expressed by polynomials with nonnegative integer coefficients.

Theorem 4. *Given a synchronization of a line of k processors in time $t(k)$, there exists a synchronization of an $(m \times n)$ array in time $t(m+n-1)$.*

Proof. An $(m \times n)$ array can be seen as many lines of $(m+n-1)$ cells, each of them having as endpoints cells $(1,1)$ and (m,n). Each of these lines corresponds to a "path" from cell $(1,1)$ to cell (m,n) going through $(m+n-3)$ other cells. Each cell (i,j) of these paths has as left neighbour either cell $(i-1,j)$ or cell $(i,j-1)$ and as right neighbour either cell $(i+1,j)$ or cell $(i,j+1)$.

Notice that the cell (i,j) is the $(i+j-1)$-th cell from the left in all the lines it belongs to. This property allows us to execute simultaneously on all these lines a synchronization in time $t(k)$ for a line of k cells. Since the length of each line is $(m+n-1)$, we have a synchronization of the $(m \times n)$ array in time $t(m+n-1)$.

In [4] synchronizations for a linear array of n cells have been given in the following times: n^2, 2^n, $n\lceil \log n \rceil$, and $n\lceil \sqrt{n} \rceil$. Using these results and the above theorem we can give the following corollary.

Corollary 1. *Given an $m \times n$ array and $K = m + n - 1$, then*

- *There are synchronizations of the $(m \times n)$ array in time K^2, 2^K, $K\lceil \log K \rceil$, and $K\lceil \sqrt{K} \rceil$.*
- *Let a and b be two integer numbers, if $aK + b \geq 2K - 1$ then there is a synchronization of the $(m \times n)$ array in time $(aK + b)$.*
- *Let $h \geq 2$ be an integer number and $a_0, \ldots, a_h$ natural numbers with $a_h \geq 1$, then there is a synchronization of the $(m \times n)$ array in time $a_h K^h + \ldots + a_1 K^1 + a_0$.*

Given a 2-CA, we can consider its time unrolling and easily extend the definitions given in section 2 to signals for 2-CA. For a 2-CA (A, C), we call Links(A, C) the set of communication links effectively used by (A, C), that is all the ordered pairs of adjacent cells x, y such that there is a bit 1 sent from x to y at some time t. We give now some results on signal composition. The first lemma says that when two signals have disjoint sets of active communication links then it is possible to obtain a new signal which is their parallel composition. The second lemma generalizes Remark 1 to the 2 dimensional case and establishes when it is possible to design a 2-CA to concatenate two signals, thus obtaining their sequential composition.

Lemma 1. *Given an $(m \times n)$ 2-CA A, let S_1 and S_2 be two signals of A on configurations C_1 and C_2, respectively. If Links$(A, C_1) \cap$ Links$(A, C_2) = \emptyset$ then there exist an $(m \times n)$ 2-CA A' and a configuration C' such that $S_1 \cup S_2$ is a signal of (A', C'). Moreover, if (A, C_1) and (A, C_2) are tailed then also (A', C') is tailed.*

Lemma 2. *Let S_1, S_2 be signals of two 2-CA's (A_1, C_1) and (A_2, C_2), respectively. The signal $cat_r(S_1, S_2)$ is the signal of a 2-CA (A, C) if the following two conditions hold:*

1. *(A_1, C_1) is tailed and (A_2, C_2) can follow (A_1, C_1);*
2. *if the site (i, j, t) belongs to the front of (A_1, C_1) and (i, j, t') belongs to the rear of (A_2, C_2), then $t < t' + r$.*

Moreover if (A_2, C_2) is tailed and Cell$(A_1, C_1) \subseteq$ Cell(A_2, C_2), then (A, C) is tailed too.

We can now give the sequential and iterated compositions of synchronizations of $(m \times n)$ arrays. In the following, if A_i is a synchronization, then G_i, L_i, and F_i are the General, Latent, and Firing states of A_i, respectively.

Theorem 5. *If A_i for $i = 1, 2$ are two synchronizations of an $(m \times n)$ array in time $t_i(m, n)$ and $d \geq 0$, then there is a synchronization in time $t_1(m, n) + t_2(m, n) + d$.*

Proof. Let S_i be the signal of (A_i, C_0), where C_0 is a standard configuration. From Lemma 2, if $r = t_1(m,n) + d$, then there exists A such that $\text{cat}_r(S_1, S_2)$ is a signal of (A, C_0). Moreover, $\text{cat}_r(S_1, S_2)$ is a synchronization in time $t(m,n) = t_1(m,n) + t_2(m,n) + d$.

Theorem 6. *If A_i for $i = 1, 2$ are two synchronizations of an $(m \times n)$ array in time $t_i(m,n)$, then there is a synchronization in time $t_1(m,n)t_2(m,n)$.*

Proof. We define a synchronization A consisting of an Iterative phase with length $t_1(m,n)$ which is executed $t_2(m,n)$ times. The set of states of A is $Q_1 \times Q_2 \times \{0,1\}^4$, the General state is $(G_1, G_2, 0, 1, 0, 0)$, the Latent state is $(L_1, L_2, 0, 0, 0, 0)$ and the Firing state is $(F_1, F_2, 0, 0, 0, 0)$. In the Iterative phase, the synchronization A modifies the first component of its state according to the transition functions of A_1, until this component is F_1. At the end of this phase A executes a transition step modifying the second component of the state according to the transition functions of A_2. Outputs of A_2 transition functions are saved in the last four components according to the order left, right, up, and down. Moreover, in this same step, A replaces F_1 with either G_1 or L_1 (depending on whether the cell is the one triggering in the initial configuration the firing signal of A_1) in the first component. So the Iterative phase can start again, until the Firing state is entered by all the cells. So, the synchronization A_1 is iterated exactly $t_2(m,n)$ times and A takes time $t_1(m,n)t_2(m,n)$.

Let A be a synchronization in time $t(m,n)$ and $X \times Y \subseteq \{1,\ldots,m\} \times \{1,\ldots,n\}$, we say that A is $(X \times Y)$-*detectable* if the states set $state(i,j,t(m,n)-1)$, $\forall (i,j) \in X \times Y$, is disjoint from the set of states $state(i',j',t(m,n)-1)$, for all $(i',j') \notin X \times Y$. Furthermore, we say that A has the *parity property* with respect to m (respectively, n), if for $i = 1,\ldots,m$ and $j = 1,\ldots,n$:

- the set of states containing $state(1,j,t(m,n)-1)$, if m is even, (respectively, $state(i,1,t(m,n)-1)$, if n is even) is disjoint from the set containing $state(1,j,t(m,n)-1)$ (respectively, $state(i,1,t(m,n)-1)$), otherwise;
- the set of states containing $state(m,j,t(m,n)-1)$, if m is even, (respectively, $state(i,n,t(m,n)-1)$, if n is even) is disjoint from the set containing $state(m,j,t(m,n)-1)$ (respectively, $state(i,n,t(m,n)-1)$), otherwise.

We recall that if we consider a line of n cells with an initial configuration $state(1,1) = state(n,1) = G$ and $state(i,1) = L$ for $i \neq 1, n$, we can synchronize the line in time n, if n is odd, and time $n-1$ otherwise. We call such a synchronization a *two-end synchronization*. Mainly, in this synchronization the linear array is split in two halves and, starting at the same time, on both the halves a minimal time synchronization is executed (the composition of the corresponding signals is possible by Lemma 1).

Lemma 3. *Let $d \geq 0$ and $m \geq d$ (respectively, $n \geq d$). Let A be a synchronization of an $(m \times n)$ array in time $t(m,n)$ with the parity property with respect to m (respectively, n) and $(X \times Y)$-detectable for a set $X \times Y = (X' \times \{1,\ldots,n\})$ (respectively, $X \times Y = (\{1,\ldots,m\} \times Y')$), where:*

- $X' = \{1, \ldots, d\} \cup \{m - d + 1, \ldots, m\} \cup \{m/2, m/2 + 1\}$ *(respectively,* $Y' = \{1, \ldots, d\} \cup \{n - d + 1, \ldots, n\} \cup \{n/2, n/2 + 1\}$*), if* m *(respectively,* n*) is even and*
- $X' = \{1, \ldots, d\} \cup \{m - d + 1, \ldots, m\} \cup \{\lceil m/2 \rceil\}$ *(respectively,* $Y' = \{1, \ldots, d\} \cup \{n - d + 1, \ldots, n\} \cup \{\lceil n/2 \rceil\}$*), otherwise.*

Then there exists a synchronization of an $(m \times n)$ *array in time* $t(m, n) + m - d$ *(respectively,* $t(m, n) + n - d$*).*

Proof. We consider only the case that A is a synchronization of an $(m \times n)$ array in time $t(m, n)$ with the parity property with respect to m, the other case is analogous. For $d = 0$, a synchronization in time $t(m, n) + m$ consists of two phases. The initial phase is the synchronization A itself and the second phase is a two-end synchronization on $(1, i), \ldots, (m, i)$ for $i = 1, \ldots, n$. By hypothesis at the end of the first phase the middle cells (or the shared middle cell, when m is odd) of each vertical line are marked, so they can behave as the first and the last in the line. Moreover, the first and the last cells are aware of the parity of m, so that they can be set in suitable General states, in such a way that the total time of the two-end synchronization is m in both cases. Thus from Lemma 2, a synchronization in time $t(m, n) + m$ exists. A synchronization A' in time $t(m, n) + m - d$ can be obtained by modifying the previous synchronization in such a way that A' jumps from the $(t(m, n) - 1)$-th configuration of A exactly to the d-th configuration of the two-end synchronization. Notice that in the d-th configuration of the two-end synchronization only the first d and the last d cells of each line are in states which are different from the Latent state. Thus, by the $(X \times Y)$-detectability of A, A' is properly defined and is a synchronization in time $t(m, n) + m - d$.

Lemma 4. *Let* A *be a synchronization of an* $(m \times n)$ *array in time* $t(m, n)$ *with the parity property with respect to* m *(respectively,* n*) and* $(X \times Y)$*-detectable for a set* $X \times Y = (X' \times \{1, \ldots, n\})$ *(respectively,* $X \times Y = (\{1, \ldots, m\} \times Y')$*) where:*

- $X' = \{m/2, m/2 + 1\}$ *(respectively,* $Y' = \{n/2, n/2 + 1\}$*) if* m *(respectively,* n*) is even, and*
- $X' = \{\lceil m/2 \rceil\}$ *(respectively,* $Y' = \{\lceil n/2 \rceil\}$*), otherwise.*

Then there is a synchronization of an $(m \times n)$ *array in time* $m\, t(m, n)$ *(respectively,* $n\, t(m, n)$*).*

Proof. We consider only the case that A is a synchronization of an $(m \times n)$ array in time $t(m, n)$ with the parity property with respect to m, the other case is analogous. We define a synchronization A' consisting of an iterative phase, with length $t(m, n)$, executed m times. The iterative phase consists of the synchronization A. The states of A' are tuples whose first component is a state of A and the second component is a state of the two-end synchronization applied to each row of the array. In the iterative phase, A' modifies the first component of

a state according to the transition functions of A. At each iteration, just a step of a two-end synchronization is performed. Thus A' is a synchronization in time $m\,t(m,n)$.

In the rest of the section we present the polynomial time synchronizations of a square array of $(n \times n)$ processors. By the minimal time synchronization presented in [5], we obtain the following result.

Remark 2. There exists a minimal time synchronization of an $(n \times n)$ square array in time $t(n) = 3n-2$ with the parity property and $(X \times Y)$-detectable for a set $X \times Y = (X' \times \{1,\ldots,n\})$, where $X' = \{n/2, n/2+1\}$, if n is even, and $X' = \{\lceil n/2 \rceil\}$, otherwise.

Thus we have the following theorems.

Theorem 7. *Let a and b be natural numbers. There is a synchronization of an $(n \times n)$ square array in time $3n-2+a(n-2)+b$.*

Proof. Directly by Remark 2, Lemma 3 (for $d=2$), and Theorem 5.

Theorem 1 shows the existence of a synchronization in time n^2, which includes a minimal time synchronization, thus the next remark follows.

Remark 3. There exists a synchronization of an $(n \times n)$ square in time n^2 with the parity property and $(X \times Y)$-detectable for a set $X \times Y = (X' \times \{1,\ldots,n\})$, where:

- $X' = \{n/2, n/2+1\}$, if n is even, and
- $X' = \{\lceil n/2 \rceil\}$, otherwise.

Finally we present synchronizations for any feasible polynomial time.

Theorem 8. *Let $h \geq 2$ be an integer number and $a_0,\ldots,a_h$ natural numbers with $a_h \geq 1$. There is a synchronization of an $(n \times n)$ array in time $a_h n^h + \ldots + a_1 n^1 + a_0$.*

Proof. From Remark 3 and Lemma 4, a synchronization in time n^b can be obtained for every $b \geq 2$. Using Theorem 5 to compose these times, the theorem follows.

References

1. K. Imai and K. Morita, *Firing squad synchronization problem in reversible cellular automata*, Theoretical Computer Science, 165 (1996), 475-482.
2. K. Kobayashy, *The Firing Squad Synchronization Problem for Two Dimensional Arrays*, Information and Control 34 (1977), 153–157.
3. S. La Torre, M. Napoli, D. Parente, *Synchronization of One-Way Connected Processors*, Complex Systems, 10 (4) (1996), pp. 239–255.
4. S. La Torre, M. Napoli, D. Parente, *Synchronization of a Line of Identical Processors at a Given Time*, Fundamenta Informaticae 34 (1998), 103-128.
5. J. Mazoyer, *On optimal solutions to the firing squad synchronization problem*, Theoretical Computer Science 168(2) (1996) 367-404.
6. E. F. Moore, *Sequential Machines, Selected Papers*, (Addison-Wesley, Reading, Mass, 1964).

P Systems with Membrane Creation: Universality and Efficiency*

Madhu Mutyam and Kamala Krithivasan

Dept. of Computer Science and Engineering
Indian Institute of Technology, Madras
Chennai-36, Tamil Nadu, India
madhu@meena.iitm.ernet.in
kamala@iitm.ernet.in

Abstract. P systems, introduced by Gh. Păun form a new class of distributed computing model. Several variants of P systems were already shown to be computationally universal. In this paper, we propose a new variant of P systems, *P systems with membrane creation*, in which some objects are productive and create membranes. This new variant of P systems is capable of solving the Hamiltonian Path Problem in linear time. We show that P systems with membrane creation are computationally complete.

1 Introduction

P systems are a class of distributed parallel computing devices of a biochemical type, introduced in [14], which can be seen as a general computing architecture where various types of objects can be processed by various operations. In the basic model one considers a membrane structure with a main membrane, called *the skin membrane*, consisting of several cell-like membranes. If a membrane does not contain any other membrane, it is called an *elementary membrane*. We formalize a membrane structure by means of well-formed parenthesized expressions, strings of correctly matching parentheses, placed in a unique pair of matching parentheses. Each pair of matching parentheses corresponds to a membrane. Graphically a membrane structure is represented by a Venn diagram without intersection and with a unique superset. The membranes delimit *regions*, where we place *objects*, elements of a finite set (an alphabet). The objects evolve according to given *evolution rules*, which are associated with regions. An object can evolve independently of the other objects in the same region of the membrane structure, or in cooperation with other objects. In particular, we can consider *catalysts*, objects which evolve only together with other objects, but are not modified by the evolution (they just help other objects to evolve). The evolution rules are given in the form of multiset transition rules, with an optional associated priority relation. The right hand side of the rules are of the form $(a, here), (a, out), (a, in_j)$,

* This research work was supported by IBM-India Research Laboratory, India.

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 276–287, 2001.

where a is an object. The meaning is that one occurrence of the symbol a is produced and remains in the same region, is sent out of the respective membrane, or is sent to membrane j (which should be reachable from the region where the rule is applied), respectively.
The application of evolution rules is done in parallel. Starting from an initial configuration (identified by the membrane structure, the objects - with multiplicities - and rules placed in its regions) and using the evolution rules, we get a *computation*. We consider a computation complete when it halts, no further rule can be applied. Two ways of assigning a result to a computation were considered: (a) by designating an internal membrane as the output membrane, (b) by reading the result outside the system. We deal here with the latter variant, where the output is obtained in the natural way: we arrange the symbols leaving the system, in the order they are expelled from the skin membrane; when several objects exit at the same time, any permutation of them is accepted.
In this paper, we propose a new variant of P systems, *P systems with membrane creation*, in which some objects are productive and create membranes. This new variant of P systems is capable of solving the Hamiltonian Path Problem in linear time. We also show that P systems with membrane creation are computationally complete.

2 Passive and Active P Systems

P systems defined in [14] have the property that their membrane structure consists of m different membranes (w.r.t the labels of membranes) and during the computation this number may decrease (by dissolving membranes) but not increase. Similarly in one of the variants of P systems, i.e., P systems with active membranes defined in [11]and [13], the membrane structure consists of m different membranes and during the computation this number may either decrease (by dissolving membranes) or increase (by dividing the existing membrane) but this number (w.r.t the labels of membranes) is always less than or equal to m. Such systems are called *passive P systems*. The degree of a passive P system is the number of initial membranes present in the system.
Sometime it may happen that different type of membranes may be increased in a membrane system by creating new membranes whose label is different from the existing ones. Such systems are called *active P systems*. Since the different type of membranes may increase during the computation, we can not take initial membranes as the degree of the membrane system. So the degree of an active P system is defined as the ordered pair consisting of the number of initial membranes as the first component and the number of different type of membranes used in the whole computation as the second component. Here the second component is always greater than or equal to the first component. Since our variant, *P systems with membrane creation* belongs to the active P systems category, we follow latter notation for describing the degree of the membrane system.

3 Language Prerequisites

A matrix grammar with appearance checking is a construct $G = (N, T, S, M, F)$, where N, T are disjoint alphabets, $S \in N$, M is a finite set of sequences of the form $(A_1 \to x_1, \cdots, A_n \to x_n), n \geq 1$, of context-free rules over $N \cup T$ (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases), and F 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 $(A_1 \to x_1, \cdots, A_n \to x_n)$ 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 $1 < i < n$, either $(1) w_i = w'_i A_i w''_i, w_{i+1} = w'_i x_i w''_i$, for some $w'_i, w''_i \in (N \cup T)^*$, or $(2) w_i = w_{i+1}, A_i$ does not appear in w_i, and some rule $A_i \to x_i$ appears in F. (The rules of a matrix are applied in order, possibly skipping the rules in F if they cannot be applied, so we say that these rules are applied in the *appearance checking* mode.)

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 $F = \phi$ (hence we do not use the appearance checking feature), the generated family is denoted by MAT.

It is known that $CF \subset MAT \subset MAT_{ac} = RE$, the inclusion being proper.

A matrix grammar $G = (N, T, S, M, F)$ 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 in 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 $X \in N_1, A \in N_2, x \in T^*$;

Moreover, there is only one matrix of type 1 and F consists exactly all rules $A \to \dagger$ appearing in matrices of type 3; $\dagger$ is called a trap symbol, because once introduced, it is never removed. A matrix of type 4 is used only once, in the last step of the derivation.

According to [6], for each matrix grammar there is an equivalent matrix grammar in the binary normal form.

For an arbitrary matrix grammar $G = (N, T, S, M, F)$, let us denote by $ac(G)$, the cardinality of the set $\{A \in N | A \to \alpha \in F\}$. From the construction in the proof of Lemma 1.3.7 in [6] one can see that if we start from a matrix grammar G and we get the grammar G' in the binary normal form, then $ac(G') = ac(G)$.

Improving the result from [12] (six nonterminals, all of them used in the appearance checking mode, suffice in order to characterize RE with matrix grammars), in [7] it was proved that four nonterminals are sufficient in order to characterize RE with matrix grammars and out of them only three are used in appearance checking rules. Of interest here is another result from [7]; if the total number of nonterminals is not restricted, then each recursively enumerable language can be generated by a matrix grammar G such that $ac(G) \leq 2$.

Consequently, to the properties of a grammar G in the binary normal form we

can add the fact that $ac(G) \leq 2$. This is called as *strong binary normal form* for matrix grammars.
A multiset over an alphabet $V = \{a_1, \cdots, a_n\}$ is a mapping μ from V to N, the set of natural numbers, and it can be represented by any string $w \in V^*$ such that $\Psi_V(w) = (\mu(a_1), \cdots, \mu(a_n))$, where Ψ_V is the Parikh mapping associated with V. Operations with multisets are defined in the natural manner.

4 P Systems with Membrane Creation

One of the primary goals of all living creatures is to survive. To do so, cells must be able to reproduce. Cells [1] can reproduce in two ways, *mitosis* and *meiosis*. Meiosis is a form of sexual reproduction and only occurs in gametes (reproductive cells). Mitosis[2] is the process in which an Eukaryotic cell divides into two equal, genetically identical parts. There are 5 stages in mitosis, those are: 1. Interphase, 2. Prophase, 3. Metaphase, 4. Anaphase, 5. Telophase. In Prophase, the nuclear membrane will dissolve and a new nuclear membrane will reform in Telophase. So in nature, it happens that membranes emerge in certain circumstances just by the organization of certain chemical compounds [15]. Similarly, each membrane (in biological sense) is having its own properties and the membranes have recognition proteins[3] with which it can recognize other membranes. With the help of *indexing* the membranes and using the target in_i we model the behavior of a membrane. We try to model the behavior of a *membrane creation* with our new variant, i.e., *P-systems with membrane creation*. Each membrane in the membrane system is having both productive and non-productive objects. A productive object is an object which can create a new membrane and transforms into other object. A non-productive object only transforms into another object without creating new membrane.
Formally a *P system with membrane creation* of degree (m, n), $n \geq m \geq 1$, is a construct

$$\Pi = (V, T, C, \mu, w_0, w_1, \cdots, w_{(m-1)}, R_0, R_1, \cdots, R_{(n-1)}),$$

where:

1. V is an alphabet; it consists of both productive and non-productive objects;
2. $T \subseteq V$, is the output alphabet;
3. $C \cap V \neq \phi$, is the set of 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 a given set; here we always use the labels 0 (for the skin membrane), $1, \cdots, (m-1)$;
5. w_i, $0 \leq i \leq (m-1)$, are multisets of objects over V associated with the regions 0, $1, \cdots, (m-1)$ of μ;
6. R_i, $0 \leq i \leq (n-1)$, are finite set of evolution rules over V. An evolution rule is of two types:
   (a) If a is a single non-productive object, then the evolution rule is in the form $a \to v$ or $ca \to cv$, where $c \in C, a \in (V - C), v = v'$ or $v = v'\delta$ or

$v = v'\tau$, where v' is a multiset of objects over $((V-C) \times \{here, out\}) \cup ((V-C) \times \{in_j | 1 \leq j \leq (n-1)\})$, and δ, τ are special symbols not in V.

(b) If a is a single productive object, then the evolution rule is in the form $a \rightarrow [_i\ v]_i$ or $ca \rightarrow c[_i\ v]_i$, where $c \in C, a \in (V-C)$, v is a multiset of objects over $(V-C)$. The rule of the form $a \rightarrow [_i\ v]_i$ means that the object identified by a is transformed into the objects identified by v, surrounded by a new membrane having the label i. No rule of the form $a \rightarrow [_0\ v]_0$ can appear in any set R_i. During a computation the number of membranes can increase or decrease.

The membrane structure and the multisets in Π constitute the *initial configuration* of the system. We can pass from a 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, should evolve simultaneously. A rule can be used only if there are objects which are *free* at the moment when we check its applicability.

The application of a rule $ca \rightarrow cv$ in a region containing a multiset w means to remove a copy of the object a in the presence of c (catalyst), providing that such copies exist, then follow the prescriptions given by v: If an object appears in v in the form $(a, here)$, then it remains in the same region; if it appears in the form (a, out), then a copy of the object a will be introduced in the region of the membrane placed outside the region of the rule $ca \rightarrow cv$; if it appears in the form (a, in_i), then a copy of a is introduced in the membrane with index i, if such a membrane exist inside the current membrane, otherwise the rule cannot be applied. If the special symbol τ appears, then the thickness of the membrane which delimits the region where we work is increased by 1. Initially, all membranes have the thickness 1. If a rule in a membrane of thickness 1 introduces the symbol τ, then the thickness of the membrane becomes 2. A membrane of thickness 2 does not become thicker by using further rules which introduce the symbol τ, but no object can enter or exit it. If a rule which introduces the symbol δ is used in a membrane (including the skin membrane) of thickness 1, then the membrane is dissolved; if the membrane had thickness 2, then it returns to thickness 1. Whenever the skin membrane is dissolved, the whole membrane system will be destroyed. If at the same step one uses rules which introduce both δ and τ in the same membrane, then the membrane does not change its thickness. No object can be communicated through a membrane of thickness two, hence rules which introduce commands *out* and in_j requesting such communications, can not be used. However, the communication has priority over changing the thickness: if at the same step an object should be communicated and a rule introduces the action τ, then the object is communicated and "after that" the membrane changes the thickness. When applying a rule $a \rightarrow [_i v]_i$ in a region j, a copy of a is removed and a membrane with the label i is created, containing the multiset v, inside the region of membrane j. We can never create a skin membrane. Similarly, when applying a rule $ca \rightarrow c[_i v]_i$ in a region j, a copy of a, in presence of c (catalyst), is removed and a membrane with the label i is created, containing the multiset v, inside the region of membrane j.

A sequence of transitions between configurations of a given P system Π is called a *computation* with respect to Π. A computation is *successful* if and only if it halts, that is, there is no rule applicable to the objects present in the last configuration. The result of a successful computation is $\Psi_T(w)$, where w describes the multiset of objects from T which have left the skin membrane during the computation. The set of such vectors $\Psi_T(w)$ is denoted by $Ps(\Pi)$ (from "Parikh set") and we say that it is *generated* by Π. (Note that we take into account only the objects from T.)
The family of all sets of vectors of natural numbers $Ps(\Pi)$ generated by a P system with membrane creation of degree $(m, n), n \geq m \geq 1$, with catalysts and the actions of both δ, τ, using the target indications of the form $here, out, in_j$, is denoted by $NPMC_{(m,n)}(Cat, tar, \delta, \tau)$; when one of the features $\alpha \in \{Cat, \delta, \tau\}$ is not present, we replace it with $n\alpha$. When the number of membranes is not bounded, we replace the subscript (m, n) with $(*, *)$.

5 Solving the Hamiltonian Path Problem for Undirected Graphs

Given an undirected graph G= (U, E) with $n(\geq 2)$ nodes, where U is the set of nodes and E, the set of edges. The Hamiltonian path problem is to determine whether or not there exist an Hamiltonian path in G, that is, to determine whether or not there exist a path that passes through all the nodes in U exactly once. The Hamiltonian Path Problem for undirected graphs is known to be NP-Complete. It is possible to solve this problem, in a time which is linear in the number of nodes of a graph, by using a P system with membrane creation.

Theorem 1. *The Hamiltonian Path Problem for undirected graphs can be solved, in a time which is linear in the number of nodes of a graph, by using a P system with membrane creation.*

Proof. Let G = (U, E) be an undirected graph with $n(\geq 2)$ nodes. Let $U = \{a_1, a_2, \cdots, a_n\}$. We now construct a P system with membrane creation of degree $(1, n+1)$, $NPMC_{(1,n+1)}(nCat, tar, \delta, \tau)$, as

$$\Pi = (V,\ T, C, \mu,\ w_0,\ R_0, R_1, \cdots, R_n),$$

where $V = \{a_i, a_i', f_i, f_i', d_i, d_i', c_i^j |\ 1 \leq i \leq n,\ 0 \leq j \leq n\} \cup \{Y\} \cup \{t_i |\ 0 \leq i \leq 3n\}$;
(here only a_i are productive objects)
$T = \{Y\}$;
$C = \phi$;
$\mu = [_0]_0$;
$w_0 = \{t_0, a_1, a_2, \cdots, a_n\}$;
The set R_0 contains the following rules:

1. $t_i \to t_{i+1}, 0 \leq i \leq 3n-1$;
2. $t_{3n} \to \delta$;
3. $c_i^j \to \lambda, 1 \leq j \leq n-1, \forall i$;

4. $c_i^n \rightarrow (Y, out), \forall i$;
5. $a_i \rightarrow [_i f_i', c_i^0, a_{j_1}, \cdots, a_{j_k}\ ]_i,\ 1 \leq i \leq n$;
   (a_i will create a new membrane with label i and transform into a multiset of objects $f_i' c_i^0 a_{j_1} \cdots a_{j_k}$, where $j_1, \cdots, j_k$ are vertices adjacent to vertex i)

The set $R_i, 1 \leq i \leq n$, contains the following rules:

1. $a_j \rightarrow [_j f_j', d_j', a_{j_1}', \cdots, a_{j_k}'\ ]_j,\ 1 \leq j \leq n$;
   (a_j will create a new membrane with label j and transform into a multiset of objects $f_j' d_j' a_{j_1}' \cdots a_{j_k}'$, where $j_1, \cdots, j_k$ are vertices adjacent to vertex j)
2. $d_i' \rightarrow d_i$;
3. $f_i' \rightarrow (f_i, in_{j_1})(f_i, in_{j_2}) \cdots (f_i, in_{j_k})$;
   (here $j_1, \cdots, j_k$ are vertices adjacent to vertex i)
4. $f_j \rightarrow (f_j, in_{j_1})(f_j, in_{j_2}) \cdots (f_j, in_{j_k}), \forall j \neq i$;
   (here $j_1, \cdots, j_k$ are vertices adjacent to vertex i)
5. $a_j' \rightarrow a_j$;
6. $d_i \rightarrow c_i^0$;
7. $f_i \rightarrow \lambda\tau$;
8. $c_j^k \rightarrow (c_j^{k+1}, out), k \geq 0$;

$$N_{(1,n+1)}(\Pi) = \begin{cases} \{Y^m, \text{ if the given graph has 'm' Hamiltonian paths.}\} \\ \phi, \quad \text{otherwise.} \end{cases}$$

The system works as follows: Initially the skin membrane is having the productive objects $a_1, \cdots, a_n$. In the skin membrane, each productive object a_i, by using the rule $a_i \rightarrow [_i f_i' c_i^0, a_{j_1}, \cdots, a_{j_k}]_i$, will create a membrane with index i and transform into a multiset of objects $f_i', c_i^0, a_{j_1}, \cdots, a_{j_k}$. In membrane i, a productive object a_j will create a membrane with index j and transform into a multiset of objects $f_j' d_j' a_{j_1}' \cdots a_{j_k}'$. In membrane i, f_j' will be replaced with f_j, d_j' will be replaced with d_j, and a_j' with a_j. In the next step, d_j will be replaced with c_j^0 and a_j will create a membrane by using the rule $a_j \rightarrow [_j f_j' d_j' a_{j_1}' \cdots a_{j_k}']_j$ and the process defined above will be repeated. Whenever a new membrane is created, all f_j's and f_i''s, from its parent membrane, will move into that membrane. The role of f_j is to increase the thickness of the membrane j, so that no object will be sent out from that membrane. This will eliminate the effect of the path, which contains multiple copies of the same node, on the result. For every rewriting step, c_j^k will be rewritten as c_j^{k+1} and sent out of the membrane i provided that the thickness of the membrane i is 1. In the skin membrane c_j^n (indicates that there exist a tour of path length n) will be replaced with Y and sent out. We use a counter t_i in the skin membrane, so that for every rewriting step it will be incremented and whenever the counter reaches t_{3n}, indicates the end of the computation, it will dissolve the whole membrane system by dissolving the skin membrane. So, if there exist a tour of length n, by traversing all nodes in the graph, then we can get Y from the system.

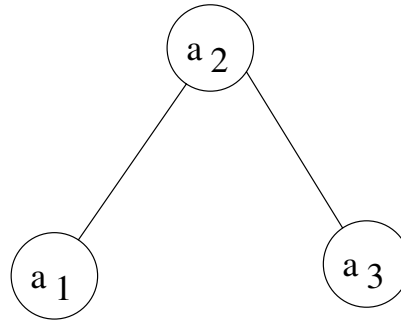
Time Complexity:

- This algorithm takes $3n + 1$ steps for generating the output.

Note:- HPP problem can also be solved using *P systems with active membranes* [11] in $3n + 1$ steps but it requires the rules for division of membranes. □

Example 1. Checking whether or not there exist an Hamiltonian Path in a given undirected graph.

Sol: Let us consider the graph $G = (\ U,\ E)$, with $U = \{a_1,\ a_2,\ a_3\}$ as shown below. We now construct a P system with membrane creation of degree $(1, 4)$ as



$$\Pi = (V, T, C, \mu, w_0, R_0, \cdots, R_3),$$

where
$V = \{a_i, a_i', f_i, f_i', d_i, d_i', c_i^j | 1 \leq i \leq 3, 0 \leq j \leq 3\} \cup \{t_i | 0 \leq i \leq 9\} \cup \{Y\}$;
$T = \{Y\}$;
$C = \phi$;
$\mu = [_0]_0$;
$w_0 = \{t_0, a_1, \cdots, a_3\}$;
and $R_i, 0 \leq i \leq 3$, as shown in the algorithm.

1. $[_0 t_0 a_1 a_2 a_3]_0$;

2. $[_0 t_1 [_1 f_1' c_1^0 a_2]_1 [_2 f_2' c_2^0 a_1 a_3]_2 [_3 f_3' c_3^0 a_2]_3]_0$;

3. $[_0 t_2 c_1^1 c_2^1 c_3^1 [_1 f_1' [_2 f_2' d_2' a_1' a_3']_2]_1 [_2 f_2' [_1 f_1' d_1' a_2']_1 [_3 f_3' d_3' a_2']_3]_2 [_3 f_3' [_2 f_2' d_2' a_1' a_3']_2]_3]_0$;

4. $[_0 t_3 [_1 [_2 f_1 f_2' d_2 a_1 a_3]_2]_1 [_2 [_1 f_1' f_2 d_1 a_2]_1 [_3 f_2 f_3' d_3 a_2]_3]_2 [_3 [_2 f_2' f_3 d_2 a_1 a_3]_2]_3]_0$;

5. $[_0 t_4 [_1 [_2 f_1 f_2' c_2^0 [_1 f_1' d_1' a_2']_1 [_3 f_3' d_3' a_2']_3]_2]_1$
$[_2 [_1 f_1' f_2 c_1^0 [_2 f_2' d_2' a_1' a_3']_2]_1 [_3 f_2 f_3' c_3^0 [_2 f_2' d_2' a_1' a_3']_2]_3]_2$
$[_3 [_2 f_2' f_3 c_2^0 [_1 f_1' d_1' a_2']_1 [_3 f_3' d_3' a_2']_3]_2]_3]_0$;

6. $[_0 t_5 [_1 c_2^1 [_2 [_1 f_1 f_2 f_1' d_1 a_2]_1 [_3 f_1 f_2 f_3' d_3 a_2]_3]_2]_1$
$[_2 c_1^1 c_3^1 [_1 [_2 f_1 f_2 f_2' d_2 a_1 a_3]_2]_1 [_3 [_2 f_2 f_3 f_2' d_2 a_1 a_3]_2]_3]_2$
$[_3 c_2^1 [_2 [_1 f_2 f_3 f_1' d_1 a_2]_1 [_3 f_2 f_3 f_3' d_3 a_2]_3]_2]_3]_0$;

7. $[_0 t_6 c_1^2 c_2^2 c_3^2 [_1 [_2 \{_1 f_2 f_1' c_1^0 [_2 f_2' d_2' a_1' a_3']_2\}_1 [_3 f_1 f_2 f_3' c_3^0 [_2 f_2' d_2' a_1' a_3']_2]_3]_2]_1$
$[_2 [_1 \{_2 f_1 f_2' c_2^0 [_1 f_1' d_1' a_2']_1 [_3 f_3' d_3' a_2']_3\}_2]_1 [_3 \{_2 f_3 f_2' c_2^0 [_1 f_1' d_1' a_2']_1 [_3 f_3' d_3' a_2']_3\}_2]_3]_2$
$[_3 [_2 [_1 f_2 f_3 f_1' c_1^0 [_2 f_2' d_2' a_1' a_3']_2]_1 \{_3 f_2 f_3' c_3^0 [_2 f_2' d_2' a_1' a_3']_2\}_3]_2]_3]_0$;

8. $[_0 t_7 [_1 [_2 c_3^1 \{_1 c_1^0 [_2 f_2 f_1 f_2' d_2 a_1 a_3]_2\}_1 [_3 [_2 f_1 f_2 f_3 f_2' d_2 a_1 a_3]_2]_3]_2]_1$
$[_2 [_1 \{_2 c_2^0 [_1 f_1 f_2 f_1' d_1 a_2]_1 [_3 f_1 f_2 f_3' d_3 a_2]_3\}_2]_1 [_3 \{_2 c_2^0 [_1 f_3 f_2 f_1' d_1 a_2]_1$
$[_3 f_3 f_2 f_3' d_3 a_2]_3\}_2]_3]_2$
$[_3 [_2 c_1^1 [_1 [_2 f_2 f_3 f_1 f_2' d_2 a_1 a_3]_2]_1 \{_3 c_3^0 [_2 f_2 f_3 f_2' d_2 a_1 a_3]_2\}_3]_2]_3]_0$;

9. $[_0 t_8 [_1 c_3^2 [_2 \{_1 c_1^0 \{_2 f_1 f_2' c_2^0 [_1 f_1' d_1' a_2']_1 [_3 f_3' d_3' a_2']_3\}_2\}_1 [_3 \{_2 f_1 f_3 f_2' c_2^0 [_1 f_1' d_1' a_2']_1$
$[_3 f_3' d_3' a_2']_3\}_2]_3]_2]_1$
$[_2 [_1 \{_2 c_2^0 \{_1 f_2 f_1' c_1^0 [_2 f_2' d_2' a_1' a_3']_2\}_1 [_3 f_1 f_2 f_3' c_3^0 [_2 f_2' d_2' a_1' a_3']_2]_3\}_2]_1$
$[_3 \{_2 c_2^0 [_1 f_3 f_2 f_1' c_1^0 [_2 f_2' d_2' a_1' a_3']_2]_1 \{_3 f_2 f_3' c_3^0 [_2 f_2' d_2' a_1' a_3']_2\}_3\}_2]_3]_2$
$[_3 c_1^2 [_2 [_1 \{_2 f_3 f_2' c_2^0 [_1 f_1' d_1' a_2']_1 [_3 f_3' d_3' a_2']_3\}_2]_1 \{_3 c_3^0 \{_2 f_3 f_2' c_2^0 [_1 f_1' d_1' a_2']_1$
$[_3 f_3' d_3' a_2']_3\}_2\}_3]_2]_3]_0$;

10. $[_0 t_9 c_1^3 c_3^3 [_1 [_2 \{_1 c_1^0 \{_2 c_2^0 [_1 f_1 f_2 f_1' d_1 a_2]_1 [_3 f_1 f_2 f_3' d_3 a_2]_3\}_2\}_1 [_3 \{_2 c_2^0 [_1 f_1 f_3 f_2 f_1' d_1 a_2]_1$
$[_3 f_1 f_3 f_2 f_3' d_3 a_2]_3\}_2]_3]_2]_1$
$[_2 [_1 \{_2 c_2^0 c_3^1 \{_1 c_1^0 [_2 f_2 f_1 f_2' d_2 a_1 a_3]_2\}_1 [_3 [_2 f_1 f_2 f_3 f_2' d_2 a_1 a_3]_2]_3\}_2]_1$
$[_3 \{_2 c_2^0 c_1^1 [_1 [_2 f_3 f_2 f_1 f_2' d_2 a_1 a_3]_2]_1 \{_3 c_3^0 [_2 f_2 f_3 f_2' d_2 a_1 a_3]_2\}_3\}_2]_3]_2$
$[_3 [_2 [_1 \{_2 c_2^0 [_1 f_3 f_2 f_1' d_1 a_2]_1 [_3 f_3 f_2 f_3' d_3 a_2]_3\}_2]_1 \{_3 c_3^0 \{_2 c_2^0 [_1 f_3 f_2 f_1' d_1 a_2]_1$
$[_3 f_3 f_2 f_3' d_3 a_2]_3\}_2\}_3]_2]_3]_0$;

11. Y^2;

Y^2 indicates that there exist 2 Hamiltonian paths in the given graph. Here $a_1 - a_2 - a_3$ and $a_3 - a_2 - a_1$ are the two paths.
Note :- Here we used different notation ({ }) for membranes of thickness 2.

6 Computational Universality

We now investigate the computational power of a P system with membrane creation. Here in *theorem 2* we show that a P system with membrane creation of degree $(1, 4)$ with catalysts and with the actions of τ and δ generate $PsRE$.

Theorem 2. $PsRE = NPMC_{(1,4)}(Cat, tar, \tau, \delta)$.

Proof. We prove only the inclusion $PsRE \subseteq NPMC_{(1,4)}(Cat, tar, \tau, \delta)$, the reverse inclusion can be proved in a straight forward manner. Let us consider a matrix grammar with appearance checking, $G = (N, T, S, M, F)$, in the strong binary normal form, that is with $N = N_1 \cup N_2 \cup \{S, \dagger\}$, with rules of the four forms mentioned in Section 3, and with $ac(G) \leq 2$. Assume that we are in the worst case, with $ac(G) = 2$, and let $B^{(1)}$ and $B^{(2)}$ be the two symbols in N_2 for which we have rules $B^{(j)} \to \dagger$ in matrices of M. Let us assume that we have h matrices of the form $m_i' : (X \to Y, B^{(j)} \to \dagger), X, Y \in N_1, j \in \{1, 2\}, 1 \leq i \leq h$, and k matrices of the form $m_i : (X \to \alpha, A \to x), X \in N_1, A \in N_2, \alpha \in N_1 \cup \{\lambda\}$,

and $x \in (N_2 \cup T)^*, 1 \leq i \leq k$. Each matrix of the form $(X \to \lambda, A \to x), X \in N_1, A \in N_2, x \in T^*$, is replaced by $(X \to f, A \to x)$, where f is a new symbol. We continue to label the obtained matrix in the same way as the original one. The matrices of the form $(X \to Y, B^{(j)} \to \dagger), X, Y \in N_1$ are labeled by m'_i, with $i \in lab_j$, for $j \in \{1, 2\}$, such that lab_1, lab_2 and $lab_0 = \{1, 2, \cdots, k\}$ are mutually disjoint sets.
We construct a *P system with membrane creation* as

$$\Pi = (V, T, C, \mu, w_0, R_0, R_1, R_2, R_3),$$

with the following components :
$V = N_1 \cup N_2 \cup \{X', X_i \mid X \in N_1, 1 \leq i \leq k\}$
$\cup \{X'_i \mid X \in N_1, 1 \leq i \leq h\}$
$\cup \{A_i | A \in N_2, 1 \leq i \leq k\}$
$\cup \{f, E, Z, \dagger\}$
$C = \{c\}$;
$\mu = [_0]_0$;
$w_0 = \{c, X, A\}$, for $(S \to XA)$ being the initial matrix of G;
and the sets R_0, R_1, R_2 and R_3 contains the following rules:

- R_0 :
  1. $X \to [_1 EX']_1$;
  2. $X \to [_{j+1} X'_i]_{j+1}$, for $m'_i : (X \to Y, B^{(j)} \to \dagger), j \in \{1, 2\}$;
  3. $cA \to c(A_1, in_0)$;
  4. $B^{(j)} \to (\dagger, in_{j+1}), j \in \{1, 2\}$;
  5. $a \to (a, out)$;
  6. $f \to \lambda$;
  7. $Z \to \lambda$;
  8. $A \to A, A \in N_2$;
- R_1 :
  1. $E \to \lambda\tau$;
  2. $X' \to X_1$;
  3. $X_i \to X_{i+1}, 1 \leq i \leq (k-1)$;
  4. $A_i \to A_{i+1}, 1 \leq i \leq (k-1)$;
  5. $X_i \to ZY\delta$, for $m_i : (X \to Y, A \to x), Y \in N_1 \cup \{f\}$;
  6. $A_i \to Zx\delta$, for $m_i : (X \to Y, A \to x), Y \in N_1 \cup \{f\}$;
  7. $Z \to \dagger$;
  8. $X_k \to \dagger$;
  9. $A_k \to \dagger$;
  10. $\dagger \to \dagger$;
- R_{j+1}, where $j \in \{1, 2\}$, such that $m'_i : (X \to Y, B^{(j)} \to \dagger)$:
  1. $X'_i \to Y$;
  2. $Y \to Y\delta$;
  3. $\dagger \to \dagger$;

The system works as follows:
Initially the skin membrane contains the objects X, A and a catalyst c. Here X is a productive object and creates a membrane. In order to simulate a matrix of type 2 or 4, the object X will create a membrane with index 0. Whenever a membrane with index 0 appears, with the help of catalyst c, a non-productive object $A \in N_2$ will move into the membrane 0 as A_1. At the same time, X' will transform into X_1 and the thickness of the membrane 0 will be increased to 2 by using the rule $E \to \lambda\tau$ (this is to prevent the duplicate copies of A to come in). Once we have X_1 and A_1 in membrane 0, the subscripts of X and A are increased step by step. At any time we can apply the rules $X_i \to ZY\delta, Y \in N_1 \cup \{f\}$ and $A_i \to Zx\delta$. But if we apply one of these rules first, then a trap symbol will be introduced by the rule $Z \to \dagger$. So, in order to get the correct simulation, we have to apply these two rules at the same step. In this way we can complete the simulation of a matrix m_i of type 2 or 4. If the symbol A in membrane 0 is not the one in the matrix $m_i : (X \to Y, A \to x)$, then we cannot apply the rule $A_i \to Zx\delta$ and a trap symbol will be introduced by the rule $Z \to \dagger$. After simulating a matrix $m_i : (X \to f, A \to x)$ of type 4, if there is any nonterminal $A \in N_2$ present in the skin membrane, then the computation never halts due to the aplication of the rule $A \to A$. The symbol f will be erased in the skin membrane.
For simulating a matrix of type 3, X will create a membrane with index $(j + 1), j \in \{1, 2\}$. If there is a symbol $B^{(j)}$ present in the skin membrane, then it will be replaced with $\dagger$ and enter into the membrane $(j + 1)$ and so that the computation never halts. Otherwise, in membrane $(j + 1)$, the symbol X'_i will be replaced with Y and in the next step, this Y will be sent out by using the rule $Y \to Y\delta$. In this way we can complete the simulation of a matrix m'_i. Thus, the equality $\Psi_T(L(G)) = NPMC_{(1,4)}(Cat, tar, \tau, \delta)$ follows. □

7 Conclusion

As a variant of P systems, we have introduced a new variant of P systems, *P systems with membrane creation*, in which some objects are productive and create membranes. This new variant of P systems is capable of solving the Hamiltonian Path Problem in linear time. We showed that P systems with membrane creation are computationally complete.

Acknowledgement

The authors wish to thank Prof. Gheorghe Păun for his comments and suggestions.

References

1. http://library.thinkquest.org/C004535/reproduction.html
2. http://fermat.stmarys-ca.edu/ jpolos/science/mitosis.html

3. http://scidiv.bcc.ctc.edu/rkr/biology101/lectures/membranes.html
4. http://bioinformatics.bio.disco.unimib.it/psystems/
5. Dassow, J., Păun, Gh.,: On the Power of Membrane Computing. Journal of Universal Computer Science, Vol. 5 (1999) 33-49.
6. Dassow, J., Păun. Gh.,: Regulated Rewriting in Formal Language Theory, Springer-Verlag, Berlin, (1989).
7. Freund, R., Păun, Gh.,: On the number of nonterminals in graph-controlled, programmed, and matrix grammars. This volume.
8. Ito, M., Martin-Vide, C., and Păun, Gh.,: A characterization of Parikh sets of ET0L languages interms of P systems, submitted (2000).
9. Krishna, S.N., Rama, R.,: On Simple P Systems with External Output. submitted (2000).
10. Krishna, S.N.,: Computing with Simple P systems. Workshop on Multiset Processing (2000).
11. Krishna, S.N., Rama, R.,: A variant of P systems with Active Membranes: Solving NP-Complete Problems. Romanian Journal of Information Science and Technology, vol. 2 (1999) 357-367.
12. Păun, Gh.,: Six nonterminals are enough for generating each r.e. language by a matrix grammar, Intern. J. Computer Math., Vol. 15 (1984), 23-37.
13. Păun, Gh.,: P systems with Active Membranes: Attacking NP-Complete problems. Journal of Automata, Languages and Combinatorics, **6**, 1 (2001).
14. Păun, Gh.,: Computing with Membranes. Journal of Computer and System Sciences. Vol. 61 (2000) 108-143.
15. Păun, Gh.,: Computing with Membranes (P systems): Twenty six Research topics. Technical Report, Auckland University (2000).
16. Rozenberg, G., Salomaa, A.,: Handbook of Formal Languages. Springer-Verlag, Berlin (1997).

On the Computational Power of a Continuous-Space Optical Model of Computation

Thomas J. Naughton and Damien Woods

TASS Group, Department of Computer Science,
National University of Ireland, Maynooth, Ireland.
dwoods@cs.may.ie

Abstract. We introduce a continuous-space model of computation. This original model is inspired by the theory of Fourier optics. We show a lower bound on the computational power of this model by Type-2 machine simulation. The limit on computational power of our model is nontrivial. We define a problem solvable with our model that is not Type-2 computable. The theory of optics does not preclude a physical implementation of our model.

1 Introduction

In this paper we introduce to the theoretical computer science community an original continuous-space model of computation. The model was developed for the analysis of (analog) Fourier optical computing architectures and algorithms, specifically pattern recognition and matrix algebra processors [3]. The functionality of the model is limited to operations routinely performed by optical scientists thus ensuring it is implementable within this physical theory [10]. The model uses a finite number of two dimensional (2-D) images of finite size and infinite resolution for data storage. It can navigate, copy, and perform other optical operations on its images. A useful analogy would be to describe the model as a random access machine, without conditional branching and with registers that hold images. This model has previously [4, 5] been shown to be at least as computationally powerful as a universal Turing machine (TM). However, its exact computational power has not yet been characterised. To demonstrate a lower bound on computational power we simulate a Type-2 machine. The upper bound is not obvious; the model can decide at least one language that a Type-2 machine can not. This combination of super-Turing power and possible implementation strongly motivates investigation of the model. In Sect. 2, we introduce the optical model of computation. In Sect. 3, we outline some relevant points from Type-2 Theory of Effectivity, and present our working view of Type-2 machines. In Sect. 4, we present our simulation of a Type-2 machine. We finish with a discussion of its super-Turing power and a conclusion (Sects. 5 and 6).

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 288–299, 2001.

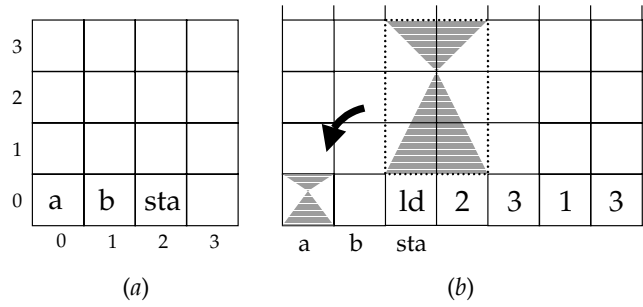


Fig. 1. Schematics of (a) the grid memory structure of our model of computation, showing example locations for the 'well-known' addresses **a**, **b** and **sta**, and (b) loading (and automatically rescaling) a subset of the grid into grid element **a**. The program | ld | 2 | 3 | 1 | 3 | . . . | *hlt* | instructs the machine to load into default location **a** the portion of the grid addressed by columns 2 through 3 and rows 1 through 3.

2 The Optical Computational Model

Each instance of our machine consists of a memory containing a program (an ordered list of operations) and an input. The memory structure is in the form of a 2-D grid of rectangular elements, as shown in Fig. 1(a). The grid has finite size and a scheme to address each element uniquely. Each grid element holds a 2-D infinite resolution complex-valued image. Three of these images are addressed by the identifiers **a**, **b**, and **sta** (two global storage locations and a program start location, respectively).

The two most basic operations available to the programmer, **ld** and **st** (both parameterised by two column addresses and two row addresses), copy rectangular $m \times n$ ($m, n \in \mathbb{N}$, $m, n \geq 1$) subsets of the grid into and out of image **a**, respectively. Upon such loading and storing the image information is rescaled to the full extent of the target location [as depicted in Fig. 1(b)]. Two additional real-valued parameters z_{lower} and z_{upper}, specifying lower and upper cut-off values, filter the rectangle's contents by amplitude before rescaling,

$$f(i,j) = \begin{cases} z_{\text{lower}} & : \ \text{Re}\,[f(i,j)] < z_{\text{lower}} \\ z_{\text{upper}} & : \ \text{Re}\,[f(i,j)] > z_{\text{upper}} \\ f(i,j) & : \ \text{otherwise} \end{cases} .$$

For the purposes of this paper we do not require the use of amplitude filtering and use an all-pass filter represented by the rationals 0/1 and 1/1 (we make use of the symbol '/' for this purpose). The complete set of atomic operations is given in Fig. 2.

Each instance of our machine is a quintuple $M = \langle D, L, Z, I, P \rangle$, in which

- $D = \langle x, y \rangle$, $x, y \in \mathbb{N}$: grid dimensions
- $L = \langle a_{\text{x}}, a_{\text{y}}, b_{\text{x}}, b_{\text{y}}, s_{\text{x}}, s_{\text{y}} \rangle$, $a, b, s \in \mathbb{N}$: locations of **a**, **b**, and **sta**
- $Z = \langle z_{\text{MIN}}, z_{\text{MAX}}, r \rangle$, $z \in \mathbb{C}$, $r \in \mathbb{Q}$: global amplitude bounds and amplitude resolution of grid elements

ld $c1$ $c2$ $r1$ $r2$ z_l z_u	: $c1, c2, r1, r2 \in \mathbb{N}$; $z_\mathrm{l}, z_\mathrm{u} \in \mathbb{Q}$; copy into **a** the rectangle defined by the coordinates ($c1$, $r1$) and ($c2$, $r2$). $(z_\mathrm{l}, z_\mathrm{u})$ is the amplitude filter [we use $(0/0, 1/1)$ everywhere for the purposes of this paper].
st $c1$ $c2$ $r1$ $r2$ z_l z_u	: $c1, c2, r1, r2 \in \mathbb{N}$; $z_\mathrm{l}, z_\mathrm{u} \in \mathbb{Q}$; copy the image in **a** into the rectangle defined by the coordinates ($c1$, $r1$) and ($c2$, $r2$).
h	: perform a horizontal 1-D Fourier transform on the 2-D image in **a**. Store result in **a**.
v	: perform a vertical 1-D Fourier transform on the 2-D image in **a**. Store result in **a**.
$\cdot$	: multiply (point by point) the two images in **a** and **b**. Store result in **a**.
$+$	: perform a complex addition of **a** and **b**. Store result in **a**.
$*$	: replace **a** with the complex conjugate of **a**.
br $c1$ $r1$	: $c1, r1 \in \mathbb{N}$; unconditionally branch to the instruction at the image with coordinates ($c1$, $r1$).
hlt	: halt.

Fig. 2. The set of atomic operations permitted in the model.

- $I = [(i_{1\mathrm{x}}, i_{1\mathrm{y}}, \psi_1), \ldots, (i_{n\mathrm{x}}, i_{n\mathrm{y}}, \psi_n)]$, $i \in \mathbb{N}$, $\psi \in Image$: the n inputs and their locations, where $Image$ is a complex surface bounded by 0 and 1 in both spatial directions and with values limited by Z
- $P = [(p_{1\mathrm{x}}, p_{1\mathrm{y}}, \pi_1), \ldots, (p_{m\mathrm{x}}, p_{m\mathrm{y}}, \pi_m)]$, $p \in \mathbb{N}$, $\pi \in \{$ld, st, h, v, $*$, $\cdot$, $+$, br, hlt, $/$, $N\} \subset Image$: the m programming symbols, for a given instance of the machine, and each of their locations. $N \in Image$ represents an arbitrary row or column address.

As might be expected for an analog processor, its programming language does not support comparison of arbitrary image values. Fortunately, not having such a comparison operator will not impede us from simulating a branching instruction (see Sect. 4). In addition, address resolution is possible since (i) our set of possible image addresses is finite (each memory grid has a fixed size), and (ii) we anticipate no false positives (we will never seek an address not from this finite set).

3 Type-2 Theory of Effectivity

Standard computability theory [8] describes a set of functions that map from one countably infinite set of finite symbol sequences to another. In "Type-2 Theory of Effectivity" (TTE) [9], 'computation' refers to processing over infinite sequences of symbols, that is, infinite input sequences are mapped to infinite output sequences. If we use two or more symbols the set of such sequences is uncountable; TTE describes computation over uncountable sets and their subsets. The following is a definition of a Type-2 machine as taken from [9].

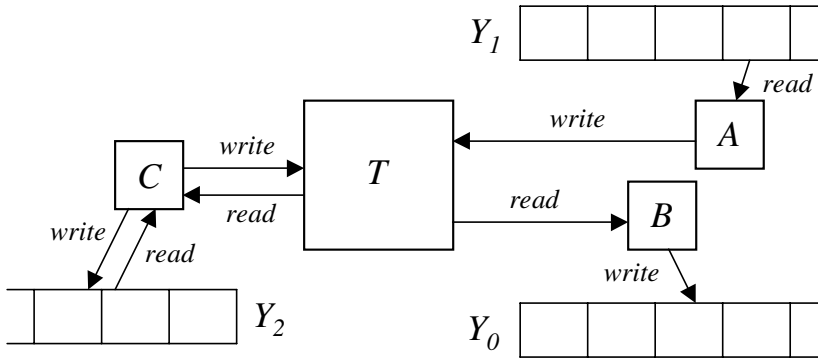


Fig. 3. Our working view of a Type-2 machine: (T) a halting TM; (Y_0) the output tape; (Y_1) the input tape; (Y_2) the nonvolatile 'work tape'. Controls A, B, and C represent the functionality to read from Y_1, write to Y_0, and read/write to Y_2, respectively.

Definition 1. *A Type-2 machine M is a TM with k input tapes together with a type specification $(Y_1, \ldots, Y_k, Y_0)$ with $Y_i \in \{\Sigma^*, \Sigma^\omega\}$, giving the type for each input tape and the output tape.*

In this definition, Σ is a finite alphabet of two or more symbols, Σ^* is the set of all finite length sequences over Σ, Σ^ω is the set of all infinite length sequences over Σ. There are two possible input/output tape types, one holds sequences from Σ^* and the other from Σ^ω.

Input tapes are one-way read only and the output tape is one-way write only. If the output tape restriction was not in place any part of an infinite output would not be guaranteed to be fixed as it could possibly be overwritten during a future computation step. Hence, finite outputs from Type-2 computations are useful for approximation or in the simulation of possibly infinite processes. A Type-2 machine either finishes its computation in finite time with a finite number of symbols on its output tape, or computes forever writing an infinite sequence. Machines that compute forever while outputting only a finite number of symbols are undefined in Type-2 theory [9].

3.1 A New View of Type-2 Computations

We maintain that a Type-2 machine can be viewed as a repeatedly instantiated halting TM that has an additional (read-only) input tape Y_1 and an additional (write-only) output tape Y_0. (Without loss of generality, the finite number of input tapes from Def. 1 can be mapped to a single tape.) A nonvolatile 'work tape' Y_2 is used to store symbols between repeated instantiations (runs) of the halting TM. This is illustrated in Fig. 3. T is the halting TM. Control A represents the functionality to read from Y_1. Control B represents the functionality to write to Y_0. Control C represents the functionality to write to and read from Y_2.

A Type-2 computation will proceed as follows. The halting TM T is instantiated at its initial state with blank internal tape(s). It reads a symbol from Y_1. Combining this with symbols (if any) left on Y_2 by the previous instantiations, it can, if required, write symbols on Y_2 and Y_0. T then halts, is instantiated once more with blank internal tape(s), and the iteration continues. The computation will either terminate with a finite sequence of symbols on the output tape or compute forever writing out an infinite sequence. In this light, an infinite Type-2 machine computation corresponds to an infinite sequence of instantiations of a single halting TM (plus extra computation steps for controls A, B, and C).

4 Simulation

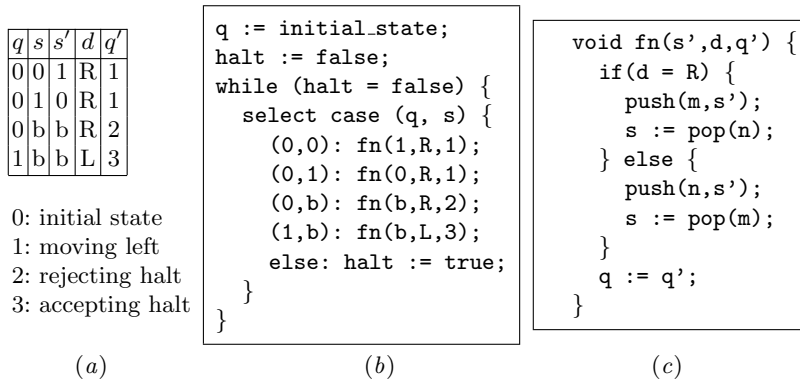
We use simulation as a technique to measure computational power. If we can show that machine M_0 can simulate every operation that machine M_1 performs, we can say that M_0 is at least as powerful as M_1. Universality for our machine has already been proved [5, 4] following Minsky's arithmetisation of TMs [2] (representing a TM in terms of quadruples of integers). Four images were used to represent Minsky's four registers. In this paper our TM simulation is more efficient, reducing the number of required commands from 52 to 17. We refine this TM simulation into a Type-2 machine simulation. Although straightforward, these simulations are technically nontrivial. We were required to overcome the restriction of no conditional branching in our model and to define an appropriate view of Type-2 computations.

In general, a TM could be simulated by a look-up table and the two stacks **m** and **n**, as shown in Fig. 4. A given TM [such as that in Fig. 4(a)] is written in the imperative form illustrated in Fig. 4(b), where the simulation of state changes and TM tape head movements can be achieved with two stacks and two variables as shown in Fig. 4(c).

In order to simulate a stack we previously effected indirect addressing with a combination of program self-modification and direct addressing. We also simulated conditional branching by combining indirect addressing and unconditional branching [5, 4]. This was based on a technique by Rojas [6] that relied on the fact that our set of symbols is finite. Without loss of generality, in our simulation we will restrict ourselves to three possible symbols, '0', '1' and a blank symbol 'b'. Then, the conditional branching instruction "if (α='1') then jump to address X, else jump to Y" is written as the unconditional branching instruction "jump to address α". We are required only to ensure that the code corresponding to addresses X and Y is always at addresses '1' and '0', respectively (and that we take into account the case where α ='b'). In a 2-D memory (with an extra addressing coordinate) many such branching instructions are possible.

4.1 Shorthand Conventions

To facilitate persons reading and writing programs, a shorthand notation is used (see Fig. 6). In this shorthand, instead of having to specify exact addresses, we



q	s	s'	d	q'
0	0	1	R	1
0	1	0	R	1
0	b	b	R	2
1	b	b	L	3

0: initial state
1: moving left
2: rejecting halt
3: accepting halt

```
q := initial_state;
halt := false;
while (halt = false) {
  select case (q, s) {
    (0,0): fn(1,R,1);
    (0,1): fn(0,R,1);
    (0,b): fn(b,R,2);
    (1,b): fn(b,L,3);
    else: halt := true;
  }
}
```

```
void fn(s',d,q') {
  if(d = R) {
    push(m,s');
    s := pop(n);
  } else {
    push(n,s');
    s := pop(m);
  }
  q := q';
}
```

(*a*) (*b*) (*c*)

Fig. 4. Figure showing (*a*) an example TM table of behaviour. This machine flips the binary value at its tape head and halts in an accepting state. If there is a blank at its tape head it halts in a rejecting state; (*b*) an illustration of how an arbitrary TM table of behaviour might be simulated with pseudocode; (*c*) how one might effect a TM computation step with stacks **m** and **n**.

give images a temporary name (such as 'x1') and refer to the address of that image with the '&' character. So, when the programmer writes 5[st|&y1|br|0|y1] (s)he would intend 5[st|13|13|5|5|0|/|1|1|/|1|br|0|] where the blank or undefined image at coordinates $(13, 5)$ will be overwritten with the contents of **a** before the statement to which it belongs is executed. Expansion from this shorthand to the long-form programming language is a mechanical procedure that could be performed as a 'tidying-up' phase by the programmer or by a pre-processor. Unless otherwise stated, we assume that the global bounds on image amplitude values are $z_{\mathrm{MIN}} = 0$ and $z_{\mathrm{MAX}} = 1$. The load and store operations contain $0/1$ $(= 0)$ and $1/1$ $(= 1)$ for their z_{lower} and z_{upper} parameters, respectively, indicating that the complete image is to be accessed. As a convention we use underlining in program grid elements whose images can be modified by the machine and italics to highlight points of TM termination within the grid.

4.2 Push and Pop Routines

Images can be stacked using a stepwise rescaling technique. Take an empty image, representing an empty stack, and an image i to be pushed onto the stack. Place both side-by-side with i to the left and rescale both into a single grid element. This could be regarded as an image stack with one element. Take another image, j, place it to the left of the stack and rescale both into a single grid element once again. This single image is a stack with two elements. A pop operation would involve stretching this stack over two grid elements. The left-hand image will then contain a single image (j in this case) and the right-hand image will contain the remainder of the stack. The stack can be repeatedly rescaled over two images popping a single image each time.

An implementation of such a stack system in our model requires indirectly addressing the stack and employing a third image (in addition to **a** and **b**) named **c**. The low-level details are as follows. In advance of the push operation we ensure that the column number of the stack is stored in **a** and the element to be pushed is stored in **c**. (The column number is sufficient as all stacks will be located on row 99 in Fig. 5.) The push routine begins by copying the contents of the stack into **b** and moving **c** into **a**. Then both **a** and **b** are rescaled into **a** and stored back in the stack's original location. In advance of a pop operation we ensure that the column number of the stack in question will be stored in **a**. The pop routine begins by loading the stack into **a** and rescaling it over both **a** and **b**. Image **a** now contains the top image and **b** contains the remainder of the stack. The top element is temporarily stored in **c**, the contents of the stack stored back in its original location, and the popped element returned to **a** before the routine ends.

We use these routines to simulate the movement of the TM tape head as illustrated in Fig. 4(c). The operational semantics of push and pop (including the use of self-modification to load the contents of a stack into **a**) can be found in rows 7 and 8 of Fig. 5.

4.3 Type-2 Machine Simulation

An arbitrary Type-2 machine is incorporated into our simulation as follows. Firstly, transform the Type-2 machine into a Type-2 machine that operates over our alphabet. Then rewrite the machine to conform to the form shown in Fig. 3. For the purposes of this simulation we represent Y_2 with TM T's internal tape (essentially using the semi-infinite tape to the left of the tape head). When T halts it will either be in an accepting or rejecting state. T's accepting state is equivalent to the simulator's initial state (i.e. T passes control back to the simulator when it halts). At the simulator's initial state it checks if T's tape head was at a non-blank symbol when T halted. If so, it writes that symbol to Y_0. All symbols to the left of the tape head (essentially the contents of Y_2) will be retained for the next instantiation of T. Next, the simulator reads a symbol from Y_1 and writes it on T's tape in the cell being scanned by T's tape head. It then passes control to T, by going into T's initial state. If at any time T halts in a rejecting state we branch to the simulator's halt state. In Fig. 5, we simulate a specific example of a Type-2 machine that flips the bits of its binary input. If the input is an infinite sequence it computes forever, writing out an infinite sequence of flipped bits. If the input is finite it outputs a finite sequence of flipped bits.

4.4 Explanation of Figs. 5 and 6

The Type-2 simulation by our model is shown in Fig. 5. It consists of two parts (separated in the diagram for clarity). The larger is the simulator (consisting of a universal TM, functionality A, B, and C from Fig. 3, and stacks Y_1 and Y_0). A TM table of behaviour must be inserted into this simulator [the example TM

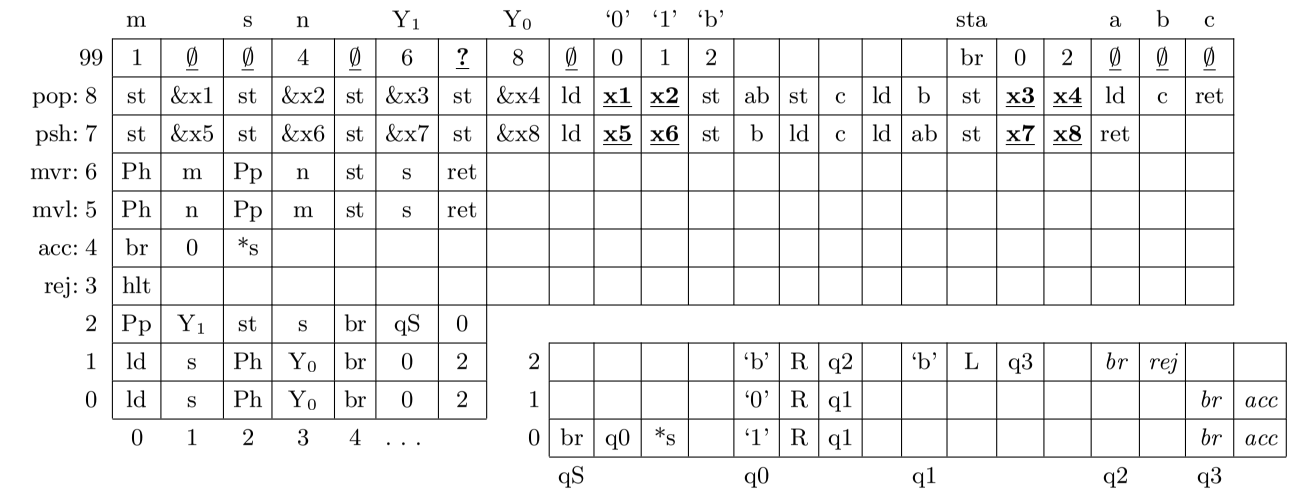


Fig. 5. Simulating Type-2 machines on our model of computation. The machine is in two parts for clarity. The larger is a universal Type-2 machine simulator and the smaller is its halting TM table of behaviour. [The example TM we use here is that in Fig. 4(a).] The simulator is written in a compact shorthand notation. The expansions into sequences of atomic operations are shown in Fig. 6 and the simulation is explained in Sect. 4.4.

(a)

Pp	Y_1	→	ld	Y_1	br	pop			
Pp	m	→	ld	m	br	pop			
Pp	n	→	ld	n	br	pop			
Ph	m	→	st	c	ld	m	br	psh	
Ph	n	→	st	c	ld	n	br	psh	
Ph	Y_0	→	st	c	ld	Y_0	br	psh	

(b)

br	q0	*s	→	ld	s	st	&y1	br	q0	**y1**
br	0	*s	→	ld	s	st	&y2	br	0	**y2**

(c)

‘b’	R	q2	→	ld	‘b’	br	mvr	br	q2	*s
‘0’	R	q1	→	ld	‘0’	br	mvr	br	q1	*s
‘1’	R	q1	→	ld	‘1’	br	mvr	br	q1	*s
‘b’	L	q3	→	ld	‘b’	br	mvl	br	q3	*s

(d)

ld	Y_1	→	ld	5	5	99	99	0	/	1	1	/	1
ld	Y_0	→	ld	7	7	99	99	0	/	1	1	/	1
ld	m	→	ld	0	0	99	99	0	/	1	1	/	1
ld	n	→	ld	3	3	99	99	0	/	1	1	/	1
st	Y_1	→	st	5	5	99	99	0	/	1	1	/	1
st	Y_0	→	st	7	7	99	99	0	/	1	1	/	1
st	m	→	st	0	0	99	99	0	/	1	1	/	1
st	n	→	st	3	3	99	99	0	/	1	1	/	1
ld	s	→	ld	2	2	99	99	0	/	1	1	/	1
ld	b	→	ld	21	21	99	99	0	/	1	1	/	1
ld	c	→	ld	22	22	99	99	0	/	1	1	/	1
st	s	→	st	2	2	99	99	0	/	1	1	/	1
st	b	→	st	21	21	99	99	0	/	1	1	/	1
st	c	→	st	22	22	99	99	0	/	1	1	/	1
ld	ab	→	ld	20	21	99	99	0	/	1	1	/	1
st	ab	→	st	20	21	99	99	0	/	1	1	/	1
ld	‘0’	→	ld	9	9	99	99	0	/	1	1	/	1
ld	‘1’	→	ld	10	10	99	99	0	/	1	1	/	1
ld	‘b’	→	ld	11	11	99	99	0	/	1	1	/	1

(e)

br	pop	→	br	0	8
br	psh	→	br	0	7
br	mvr	→	br	0	6
br	mvl	→	br	0	5
br	*acc*	→	br	0	4
br	*rej*	→	br	0	3

(f)

st	&x1	→	st	9	9	8	8	0	/	1	1	/	1	
ld	**x1**	**x2**	→	ld	**x1**	**x2**	99	99	0	/	1	1	/	1
st	**x3**	**x4**	→	st	**x3**	**x4**	99	99	0	/	1	1	/	1

Fig. 6. Time-saving shorthand conventions used in the simulator in Fig. 5. These are explained in Sect. 4.4.

is from Fig. 4(a)]. It has a straightforward encoding. Notice how for each row of the table of behaviour $\langle q, s, s', d, q' \rangle$ an ordered triple $\langle s', d, q' \rangle$ is placed at the location addressed by coordinates (q, s).

Row 99 of Fig. 5 is the 'well-known' row containing stacks, constants and the locations **a**, **b**, **c**, and **sta**. To permit indirect addressing, locations **m**, **n**, Y_1, and Y_0 store the column numbers of their respective stacks. Y_1 and Y_0 represent the one-way tapes from Fig. 3 (we pop from Y_1 and push to Y_0). The stack **m** encodes all symbols on T's tape to the left of tape head, and the stack **n** encodes all symbols on T's tape to the right of the tape head. Image **s** encodes the symbol currently being scanned by T's tape head. The blank constant 'b' is represented by '2'. Before execution begins, an input will have been encoded in stack Y_1, grid element $(6, 99)$, in place of symbol '**?**'. Control flow begins at the program start location **sta**, grid element $(17, 99)$, and proceeds rightwards until one of **br** or **hlt** is encountered.

In rows 7 and 8 we have the push and pop routines; these are explained in Sect. 4.2. Rows 5 and 6, named **mvl** and **mvr** respectively, contain commands to simulate moving left and right on T's tape. This is effected by a combination of pushes and pops to the stacks **m** and **n** and image **s** that encode T's tape symbols. The 'ret' command returns execution to the point from which a subroutine **mvl**, **mvr**, **psh**, or **pop** was called. Rows 3 and 4 describe what is to happen when the inserted TM halts in a rejecting or accepting state, respectively. In the former case the Type-2 machine simulation halts. In the latter case, the TM is reinstantiated in accordance with the description in Sect. 3.1. If the TM's tape head had written '0' or '1' as it halted, control flow then moves to row 0 or 1, respectively, and the appropriate symbol is written to Y_0 before instantiation. If the tape head writes a '2' (a blank) as it halted, control flow moves to row 2 and the TM is directly reinstantiated.

For our particular example inserted TM [from Fig. 4(a)] if it reads a '0' or '1' it halts in an accepting state and control moves to the beginning of row 4. If the input to our simulation is finite, an instantiation will eventually read a blank symbol, will enter state 'q2', and the Type-2 simulation will halt. Otherwise, it will be repeatedly instantiated, each time flipping one bit.

The simulation is written in a shorthand notation (including shorthand versions of **ld**, **st**, and **br** from Fig. 2) which is expanded using Fig. 6. Figure 6(a) shows the expansion of shorthand notation used in setting up calls to the **psh** and **pop** routines (loading the appropriate stack address into **a** in advance of a pop, and storing into **c** the symbol to be pushed and loading the stack address into **a** in advance of a push). In advance of a **psh** the element to be pushed will be in **a**. After a **pop**, the popped element will be in **a**. Figure 6(b) shows commands for branching to an address where the row is specified by the symbol currently scanned by T's tape head. Figure 6(c) shows routines for simulating the execution of a row of T's table of behaviour. Commands for loading from and storing to locations specified at runtime, and to/from the 'well-known' locations on row 99 are in Fig. 6(d). Figure 6(e) shows the commands for branching to subroutines. Finally, Fig. 6(f) illustrates how all labels are eventually given

absolute addresses by a preprocessor. After a first pass of the preprocessor (expanding the shorthand) the modifiable (underlined) references are updated with hardcoded addresses.

5 Super-Turing Capabilities of Our Model

Type-2 machines do not describe all of the computational capabilities of our model. The model's atomic operations operate on a continuum of values in constant time (independent of input size) and would not have obvious TM or Type-2 machine implementations.

Consider the language L defined by its characteristic function $f : \Sigma^\omega \to \{0, 1\}$, where

$$f(p) := \begin{cases} 1 & : \text{ if } p \neq 0^\omega \\ 0 & : \text{ otherwise} \end{cases}$$

and where p is an infinite sequence over alphabet $\{0, 1\}$. This language is acceptable but not decidable by a Type-2 machine [9] (Ex. 2.1.4.6). In our model, we encode a boolean value in an image by letting a δ-function at its origin denote a '1' and an empty image (or an image with low background noise) denote a '0'. An infinite sequence of boolean-valued images could be presented as input concatenated together in one image without loss of information (by definition, images in our machine have infinite spatial resolution). An off-centre peak can be centred for easy detection through Fourier transformation (using the shorthand program $\boxed{\text{ld}}\boxed{\text{Y}_1}\boxed{\text{h}}\boxed{\text{v}}\boxed{\text{st}}\boxed{\text{b}}\boxed{*}\boxed{\cdot}$). This uses the property that the term at the origin of a Fourier transform of an image, the dc term, has a value proportional to the energy over the entire image. Our model could therefore Fourier transform the continuous input image and then measure the value of the dc term in unit time. A peak would indicate that there is some energy (and therefore at least one '1') somewhere in the image; the corresponding word is in L. An absence of a peak at the origin indicates that there is not a '1' in the image; the corresponding word is not in L.

6 Conclusion

We introduce an original continuous-space model of computation to the computer science community. We show its relationship to existing theory by proving that it is at least as powerful as the Type-2 machine model. We are currently investigating the relationship between our model and piecewise affine maps [1, 7]. As the model remains faithful to the theory of physical optics we propose that it could serve as an implementation of Type-2 machines. Furthermore, the upper bound on the computational power of this model is nontrivial and deserves to be analysed. We present at least one problem, decidable by our model, that is undecidable by a Type-2 machine. The model's computational power, combined with implementation possibilities, strongly motivates continued investigation.

Acknowledgements

We gratefully acknowledge advice and assistance from J. Paul Gibson and the Theoretical Aspects of Software Systems research group, NUI Maynooth. Many thanks also to the reviewers of this paper for their constructive comments.

References

1. P. Koiran, M. Cosnard, and M. Garzon. Computability with low-dimensional dynamical systems. *Theoretical Computer Science*, 132(1-2):113–128, 1994.
2. M.L. Minsky. *Computation: Finite and Infinite Machines*. Series in Automatic Computation. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
3. T. Naughton, Z. Javadpour, J. Keating, M. Klíma, and J. Rott. General-purpose acousto-optic connectionist processor. *Optical Engineering*, 38(7):1170–1177, July 1999.
4. T.J. Naughton. A model of computation for Fourier optical processors. In *Optics in Computing 2000*, Proc. SPIE vol. 4089, pp. 24–34, Quebec, June 2000.
5. T.J. Naughton. Continuous-space model of computation is Turing universal. In *Critical Technologies for the Future of Computing*, Proc. SPIE vol. 4109, pp. 121–128, San Diego, California, August 2000.
6. R. Rojas. Conditional branching is not necessary for universal computation in von Neumann computers. *J. Universal Computer Science*, 2(11):756–768, 1996.
7. H.T. Siegelmann and E.D. Sontag. On the computational power of neural nets. In *Proc. 5th Annual ACM Workshop on Computational Learning Theory*, pp. 440–449, Pittsburgh, July 1992.
8. A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society ser. 2*, 42(2):230–265, 1936. Correction in vol. 43, pp. 544–546, 1937.
9. K. Weihrauch. *Computable Analysis*. Springer, Berlin, 2000.
10. D. Woods and J.P. Gibson. On the relationship between computational models and scientific theories. Technical Report NUIM-CS-TR-2001-05, National University of Ireland, Maynooth, Ireland, February 2001.

On a P-optimal Proof System for the Set of All Satisfiable Boolean Formulas (SAT)

Zenon Sadowski

Institute of Mathematics,
University of Białystok
15-267 Białystok, ul. Akademicka 2, Poland
sadowski@math.uwb.edu.pl

Abstract. In this paper we show that the problem of the existence of a p-optimal proof system for SAT can be characterized in a similar manner as J. Hartmanis and L. Hemachandra characterized the problem of the existence of complete languages for UP. Namely, there exists a p-optimal proof system for SAT if and only if there is a suitable recursive presentation of the class of all easy (polynomial time recognizable) subsets of SAT. Using this characterization we prove that if there does not exist a p-optimal proof system for SAT, then for every theory T there exists an easy subset of SAT which is not T-provably easy.

1 Introduction

The question of the existence of a p-optimal proof system for SAT (the set of all satisfiable boolean formulas) was posed by J. Köbler and J. Messner in [6]. This question, like the similar and older problem of the existence of a p-optimal proof system for $TAUT$ (the set of all tautologies in Propositional Logic), is still open. The related problem of whether the natural proof system for SAT (a satisfying truth assignment is a proof of formula α) is p-optimal, was considered in different nomenclature by S. Fenner, L. Fortnow, A. Naik and J. Rogers in [3] (see also [7]).

A proof system for a language L is a polynomial time computable function whose range is L. This notion was defined by S. Cook and R. Reckhow in [2]. To compare the efficiency of different proof systems, they considered the notion of p-simulation. Intuitively a proof system h p-simulates a second one g if there is a polynomial time computable function t translating proofs in g into proofs in h. A proof system is called p-optimal for L when it p-simulates any proof system for L.

The existence of a p-optimal proof system for $TAUT$, historically the first question about optimal proof systems, was posed by J. Krajíček and P. Pudlák [8] in 1989. Since then, research concerning optimal proof systems has acquired importance in Proof Complexity Theory and Structural Complexity Theory. J. Köbler and J. Messner [6] proved that the existence of both p-optimal proof systems for SAT and for $TAUT$ suffices to obtain a complete language for

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 300–307, 2001.

NP ∩ **co-NP**. This result generated interest in the problem of the existence of a p-optimal proof system for SAT .

It is not currently known whether **UP** and other promise classes have complete languages. J. Hartmanis, and L. Hemachandra pointed out in [4] that **UP** possesses complete languages if and only if there is a recursive enumeration of polynomial time clocked Turing machines covering all languages from this class. Earlier, the question of whether **NP** ∩ **co-NP** possesses complete languages was related to an analogous statement (see [5]).

In this paper we show that the question of the existence of a p-optimal proof system for SAT can be characterized in a similar manner. We prove that a p-optimal proof system for SAT exists if and only if there is a recursive enumeration of polynomial time clocked Turing machines covering all easy (polynomial time recognizable) subsets of SAT. This means that the problems of the existence of complete languages for promise classes and the problem of the existence of a p-optimal proof system for SAT, though distant at first sight, are structurally similar. Since complete languages for promise classes have been unsuccesfully searched for in the past, our equivalence gives some evidence of the fact that p-optimal proof systems for SAT might not exist.

2 Preliminaries

We assume some familiarity with basic complexity theory, see [1]. The symbol Σ denotes, throughout the paper, a certain fixed finite alphabet. The set of all strings over Σ is denoted by $\Sigma^\star$. For a string x, $|x|$ denotes the length of x. The symbol FP denotes the class of functions that can be computed in polynomial time.

We use Turing machines (acceptors and transducers) as our basic computational model. We will not distinguish between a machine and its code. For a deterministic Turing machine M and an input w, TIME(M, w) denotes the computing time of M on w.

We consider polynomial time clocked Turing machines with uniformly attached standard $n^k + k$ clocks which stop their computations in polynomial time (see [1]). We impose some restrictions on our encoding of these machines. From the code of any polynomial time clocked Turing machine, we can detect easily (in polynomial time) the natural k such that $n^k + k$ is its polynomial time bound.

The symbol $TAUT$ denotes the set (of all encodings) of propositional tautologies; SAT denotes the set of all satisfiable boolean formulas.

Finally, $< .,\ldots,. >$ denotes some standard polynomial time computable tupling function.

3 P-optimal Proof Systems and Almost Optimal Algorithms for SAT

The systematic study of the efficiency of propositional proof systems (proof systems for $TAUT$) was started by S. Cook and R. Reckhow in [2]. They introduced

the abstract notion of a proof system for $TAUT$ and proved that a polynomially bounded proof system for $TAUT$ exists if and only if **NP=co-NP** (see [2,10]).

The notion of a proof system can be considered not only for $TAUT$ but also for any language L.

Definition 1. *(see [2]) A proof system for L is a polynomial time computable function $h : \Sigma^\star \stackrel{onto}{\longrightarrow} L$.*

If $h(w) = x$ we may say that w is a proof of x in h. It follows from the above definition that for any proof system h there exists a polynomial time clocked transducer M_h which computes h.

To classify propositional proof systems by their relative efficiency S. Cook and R. Reckhow introduced in [2] the notion of p-simulation. For proof systems for SAT this notion can be defined as follows:

Definition 2. *(see [2]) Let h, h' be two proof systems for SAT. We say that h p-simulates h' if there exists a polynomial time computable function $\gamma : \Sigma^\star \longrightarrow \Sigma^\star$ such that for every $\alpha \in SAT$ and every $w \in \Sigma^\star$, if w is a proof of α in h', then $\gamma(w)$ is a proof of α in h.*

Definition 3. *(see [8]) A proof system for SAT is p-optimal if it p-simulates any proof system for SAT.*

The notion of an almost optimal deterministic algorithm for $TAUT$ was introduced by J. Krajíček and P. Pudlák [8]. We will consider an analogous notion of an almost optimal deterministic algorithm for SAT with Krajíček-Pudlák's optimality property.

Definition 4. *(cf. [11]) An almost optimal deterministic algorithm for SAT is a deterministic Turing machine M which recognizes SAT and such that for any deterministic Turing machine M' which recognizes SAT there exists a polynomial p such that for every satisfiable boolean formula α*

$$TIME(M;\alpha) \leq p(|\alpha|, TIME(M';\alpha))$$

We name this algorithm as an almost optimal because the optimality condition is stated only for any input string x which belongs to SAT and nothing is claimed for other $x's$ (compare the definition of an optimal acceptor for SAT in [9]).

The following result was proved in [11] (see also [9]).

Theorem 1. *Statements (i) - (ii) are equivalent.*

(i) There exists an almost optimal deterministic algorithm for SAT.
(ii) There exists a p-optimal proof system for SAT.

4 P-optimal Proof System for SAT and the Structure of Easy Subsets of SAT

The problems of the existence of complete languages for **NP** ∩ **co-NP** and **UP** are oracle dependent (see [12] and [4]), and therefore they seem to be very difficult. It was shown in [5] and [4] that these problems can be related to statements about the existence of recursive enumerations of polynomial time clocked Turing machines covering all languages from these classes. We will show in this section that the problem of the existence of a p-optimal proof system for SAT can also be related to an analogous statement.

Definition 5. *We say that a polynomial time clocked transducer M behaves well on input w if M on w outputs a satisfiable boolean formula.*

To any polynomial time clocked transducer M and any $w \in \Sigma^\star$ we shall assign the boolean formula $TEST_{M,w}$ such that:
The formula $TEST_{M,w}$ is satisfiable if and only if M behaves well on input w.

Our construction of the formula $TEST_{M,w}$ is adapted from Cook's proof that SAT is **NP**-complete. Let S be a fixed nondeterministic Turing machine working in polynomial time and accepting SAT. Let M' be the Turing machine which on any input x runs M and then runs S on the output produced by M. The formula $TEST_{M,w}$ is just Cook's formula for the pair $< M', w >$.

Test formulas possess the following properties:

(1) Global uniformity property
There exists a function $f \in FP$ such that for any polynomial time clocked transducer N with time bound $n^k + k$ and for any $w \in \Sigma^\star$

$$f(\langle N, w, 0^{|w|^k+k}\rangle) = TEST_{N,w}$$

(2) Local uniformity property
Let M be any fixed polynomial time clocked transducer. There exists a function $f_M \in FP$ such that for any $w \in \Sigma^\star$

$$f_M(w) = TEST_{M,w}$$

Definition 6. *By an easy subset of SAT we mean a set B such that $B \subset SAT$ and $B \in \mathbf{P}$.*

Let $D_1, D_2, D_3, ...$ denote the standard enumeration of all polynomial time clocked deterministic Turing machines.

Theorem 2. *Statements (i) - (ii) are equivalent.*

(i) There exists a p-optimal proof system for SAT.
(ii) The class of all easy subsets of SAT possesses a recursive $\mathbf{P}$-presentation.

By the statement (ii) we mean: there exists a recursively enumerable list of deterministic polynomial time clocked Turing machines $D_{i_1}, D_{i_2}, D_{i_3}, ...$ such that

(1) $L(D_{i_j}) \subset SAT$ for every j
(2) For every $A \subset SAT$ such that $A \in \mathbf{P}$ there exists j such that $A = L(D_{i_j})$

Proof. $(i) \rightarrow (ii)$ The existence of a p-optimal proof system for SAT is equivalent to the existence of an almost optimal deterministic algorithm for SAT. Let M be such an algorithm. A recursive $\mathbf{P}$-presentation of all easy subsets of SAT we will define in two steps. In the first step we define a recursively enumerable list of deterministic Turing machines F_1, F_2, F_3,... The machine F_k is obtained by attaching the shut-off clock $n^k + k$ to the machine M. On any input w, the machine F_k accepts w if and only if M accepts w in no more than $n^k + k$ steps, where $n = |w|$. The sequence F_1, F_2, F_3, F_4, ... of deterministic Turing machines possesses the properties (1) and (2):

(1) For every i it holds $L(F_i) \subset SAT$
(2) For every A which is an easy subset of SAT there exists j such that $A \subset L(F_j)$

To prove (2) let us consider A, an easy subset of SAT recognized by a Turing machine M'' working in polynomial time. Combining this machine with the "brute force" algorithm for SAT we obtain the deterministic Turing machine M' recognizing SAT. From the definition of M' it follows that there exists a polynomial p such that for any $\alpha \in A$, $TIME(M'; \alpha) \leq p(|\alpha|)$. Since M is an almost optimal deterministic algorithm for SAT there exists a polynomial q such that for any $\alpha \in A$, $TIME(M; \alpha) \leq q(|\alpha|)$. From this we conclude that for a sufficiently large j, $TIME(M; \alpha) \leq |\alpha|^j + j$ for any $\alpha \in A$, hence $A \subset L(F_j)$.

In the second step we define the new recursively enumerable list of deterministic polynomial time clocked Turing machines K_1, K_2, K_3,... We define K_n, $n = \langle i, j \rangle$, as the machine which simulates $n^i + i$ steps of F_i and $n^j + j$ steps of D_j and accepts w if and only if both F_i and D_j accept w and rejects w in the opposite case.

Let A be any fixed easy subset of SAT. There exist k and m such that $A = L(D_k)$ and $A \subset L(F_m)$, hence A is recognized by the machine $K_{\langle m,k \rangle}$. From this we conclude that K_1, K_2, K_3, ... provides a recursive $\mathbf{P}$-presentation of all easy subsets of SAT.

$(ii) \rightarrow (i)$

Let G be the machine generating the codes of the machines from the sequence D_{i_1}, D_{i_2}, D_{i_3},... forming a recursive $\mathbf{P}$-presentation of all easy subsets of SAT. We say that a string $v \in \Sigma^\star$ is in good form if

$$v = \langle M, w, Comp - G, Comp - TEST_{M,w}, 0^{|w|^k + k} \rangle$$

where:

M is a polynomial time clocked Turing transducer with $n^k + k$ time bound,

$w \in \Sigma^\star$,
$Comp - G$ is a computation of the machine G. This computation produces a code of a certain machine D_{i_j},
$Comp{-}TEST_{M,w}$ is a computation of the machine D_{i_j} accepting the formula $TEST_{M,w}$,
$0^{|w|^k+k}$ is the sequence of zeros (padding).

Let us note, that if v is in good form then $TEST_{M,w}$ is satisfiable as a formula accepted by a certain machine from **P**-presentation. This clearly forces M to behave well on input w, so M on input w produces a satisfiable boolean formula.

Let α_0 be a certain fixed satisfiable boolean formula. We define $Opt : \Sigma^\star \longrightarrow \Sigma^\star$ in the following way: $Opt(v) = \alpha$ if v is in good form

$$(v = \langle M, w, Comp - G, Comp - TEST_{M,w}, 0^{|w|^k+k}\rangle)$$

and α is a satisfiable boolean formula produced by M on input w, otherwise $Opt(v) = \alpha_0$.

Clearly, $Opt : \Sigma^\star \stackrel{onto}{\longrightarrow} SAT$.

In order to prove that Opt is polynomial time computable it is sufficient to notice that using global uniformity property we can check in polynomial time whether v is in good form. Hence Opt is a proof system for SAT.

It remains to be proved that Opt p-simulates any proof system for SAT. Let h be a proof system for SAT computed by a polynomial time clocked transducer K with time bound $n^l + l$. From the structure of Cook's reduction (as $TEST_{K,w}$ clearly displays K and w) it follows that the set $A_K = \{TEST_{K,w}\colon w \in \Sigma^\star\}$ is an easy subset of SAT. Therefore there exists the machine D_{i_j} from the **P**-presentation such that $A_K = L(D_{i_j})$. The function $t : \Sigma^\star \longrightarrow \Sigma^\star$ translating proofs in h into proofs in Opt can be defined in the following way:

$$t(x) = \langle K, x, Comp - G, Comp - TEST_{K,x}, 0^{|x|^l+l}\rangle$$

The word $Comp - G$ in the definition of t is the computation of G producing the code of D_{i_j}, $Comp - TEST_{K,x}$ is a computation of D_{i_j} accepting $TEST_{K,x}$.

In order to prove that t is polynomial time computable let us notice that $Comp - G$ is fixed and connected with K, so the only task is to prove that $Comp - TEST_{K,x}$ can be constructed in polynomial time. This follows from the local uniformity property of $TEST_{K,x}$ formulas and from the fact that D_{i_j} is deterministic and works in polynomial time.

5 Independence Result

Let T be any formal theory whose language contains the language of arithmetic, i. e. the language $\{0, 1, \leq, =, +, \cdot\}$. We will not specify T in detail but only assume that T is sound (that is, in T we can prove only true theorems) and the set of all theorems of T is recursively enumerable.

The notation $T \vdash \beta$ means that a first order formula β is provable in T.
Let M be a deterministic Turing machine.

Definition 7. *We say that M is consistent if M accepts only satisfiable boolean formulas (if M accepts w, then $w \in SAT$).*

By "$L(M) \subset SAT$" we denote the first order formula which expresses the consistency of M , i.e. $\forall_{w \in L(M)}[w$ is a satisfiable boolean formula].

Definition 8. *A deterministic Turing machine M is $T-$consistent if and only if $T \vdash$ "$L(M) \subset SAT$".*

Definition 9. *A set $A \subset SAT$ is T-provably easy if there exists deterministic polynomial time clocked Turing machine M fulfilling (1) – (2)*

(1) M is $T-$consistent;
(2) $L(M)$=A.

Theorem 2 can be exploited to obtain the following independence result.

Theorem 3. *If there does not exist a p-optimal proof system for SAT, then for every theory T there exists an easy subset of SAT which is not T-provably easy.*

Proof. Suppose, on the contrary, that there exists a theory T such that all easy subsets of SAT are T-provably easy. Then the following recursively enumerable set of machines Ω_T=$\{M$: M is a deterministic polynomial time clocked Turing machine which is $T-$*consistent*$\}$ creates a recursive **P**-presentation of the class of all easy subsets of SAT. By Theorem 2, this implies that there exists a p-optimal proof system for SAT, giving a contradiction.

References

1. Balcazar, J.L., Díaz, J., Gabarró, J.: Structural complexity I. 2nd edn. Springer-Verlag, Berlin Heidelberg New York (1995)
2. Cook, S.A., Reckhow R.A.: The relative efficiency of propositional proof systems. J. Symbolic Logic 44 (1979) 36–50
3. Fenner, S., Fortnow, L., Naik, A., Rogers, R.: On inverting onto functions. In: Proc. 11th Annual IEEE Conference on Computational Complexity, (1996) 213 – 222
4. Hartmanis, J., Hemachandra, L.: Complexity classes without machines: On complete languages for UP. Theoret. Comput. Sci. 58 (1988) 129 – 142
5. Kowalczyk, W.: Some connections between presentability of complexity classes and the power of formal systems of reasoning. In: Proc. Mathematical Foundations of Computer Science. Lecture Notes in Computer Science, Vol. 176. Springer-Verlag, Berlin Heidelberg New York (1988) 364 – 369
6. Köbler, J., Messner, J.: Complete problems for promise classes by optimal proof systems for test sets. In: Proc. 13th Annual IEEE Conference on Computational Complexity, (1998) 132–140

7. Köbler, J., Messner, J.: Is the standard proof system for SAT p-optimal? In: Proc 20th Annual Conference on the Foundations of Software Technology and Theoretical Computer Science. Lecture Notes in Computer Science, Vol. 1974. Springer-Verlag, Berlin Heidelberg New York (2000) 361 – 372
8. Krajíček, J., Pudlák, P.: Propositional proof systems, the consistency of first order theories and the complexity of computations. J. Symbolic Logic 54 (1989) 1063–1079
9. Messner,J.: On optimal algorithms and optimal proof systems. In: Proc. 16th Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science, Vol. 1563. Springer-Verlag, Berlin Heidelberg New York (1999) 541 –550
10. Messner, J., Torán, J.: Optimal proof systems for Propositional Logic and complete sets. In: Proc. 15th Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science, Vol. 1373. Springer-Verlag, Berlin Heidelberg New York (1998) 477 – 487
11. Sadowski, Z.: On an optimal deterministic algorithm for SAT. In: Proc 12th International Workshop, CSL'98. Lecture Notes in Computer Science, Vol. 1584. Springer-Verlag, Berlin Heidelberg New York (1999) 179 – 187
12. Sipser, M.: On relativization and the existence of complete sets. In: Proc. Internat. Coll. on Automata, Languages and Programming. Lecture Notes in Computer Science, Vol. 140. Springer-Verlag, Berlin Heidelberg New York (1982) 523 –531

D0L System + Watson-Crick Complementarity = Universal Computation

Petr Sosík

Institute of Computer Science, Silesian University,
746 01 Opava, Czech Republic,
petr.sosik@fpf.slu.cz

Abstract. D0L systems consist of iterated letter-to-string morphism over a finite alphabet. Their descriptional power is rather limited and their length sequences can be expressed as sum of products of polynomial and exponential functions. There were several attempts to enrich their structure by various types of regulation, see e.g. [1], leading to more powerful mechanisms.
Due to increasing interest in biocomputing models, V. Mihalache and A. Salomaa suggested in 1997 Watson-Crick D0L systems with so called Watson-Crick morphism. This letter-to-letter morphism maps a letter to the complementary letter, similarly as a nucleotide is joined with the complementary one during transfer of genetic information. Moreover, this new morphism is triggered by a simple condition (called trigger), e.g. with majority of pyrimidines over purines in a string.
This paper deals with the expressive power of standard Watson-Crick D0L systems. A rather unexpected result is obtained: any Turing computable function can be computed by a Watson-Crick D0L system.

1 Watson-Crick D0L Schemes

For elements of formal language theory we refer to [7,10]. Here we only briefly fix some notation. For a finite alphabet Σ, denote $(\Sigma, \cdot)$ a free monoid with the catenation operation and the empty word λ. For $a \in \Sigma$, $w \in \Sigma^*$, $|w|_a$ is the number of occurrences of a in w. For $\Gamma \subseteq \Sigma$, $|w|_\Gamma = \sum_{a\in\Gamma} |w|_a$. For $w \in \Sigma^*$ we denote w^n the catenation of n copies of w for $n \geq 1$.

A concept of Watson-Crick alphabet is a straightforward generalization of notions of "natural" DNA alphabet consisting of symbols $\{A, C, G, T\}$.

A *DNA-like alphabet* Σ is an alphabet with an even cardinality $2n$, $n \geq 1$, where the letters are enumerated as follows:

$$\Sigma = \{a_1, \ldots, a_n, \overline{a_1}, \ldots, \overline{a_n}\}.$$

We say that a_i and $\overline{a_i}$ are *complementary letters.* The letter to letter endomorphism h_W of Σ^* mapping each letter to the complementary letter is called the *Watson-Crick morphism.* Hence

$$h_W(a_i) = \overline{a_i},\ h_w(\overline{a_i}) = a_i, \quad 1 \leq i \leq n.$$

M. Margenstern and Y. Rogozhin (Eds.): MCU 2001, LNCS 2055, pp. 308–319, 2001.

In analogy with the DNA alphabet we call the non-barred letters *purines* and the barred letters *pyrimidines.* The subset of Σ^* consisting of all words, where the number of occurrences of pyrimidines is strictly greater than that of purines is denoted by PYR. The complement of PYR is denoted by PUR. We further denote the sets $\Sigma^{\mathrm{PUR}} = \{a_1, \ldots, a_n\}$ and $\Sigma^{\mathrm{PYR}} = \{\overline{a_1}, \ldots, \overline{a_n}\}$. For a set $\Delta \subseteq \Sigma$, we denote $h_W(\Delta) = \{h_W(a) \mid a \in \Delta\}$.

Definition 1. *A* standard Watson-Crick D0L scheme *(or shortly* scheme*) is a construct $G = (\Sigma, p)$, where $\Sigma = \{a_1, \ldots, a_n, \overline{a_1}, \ldots, \overline{a_n}\}$, $p : \Sigma^* \longrightarrow \Sigma^*$ is a morphism. Given a word $w_0 \in \Sigma^*$, the* derivation sequence $S(G, w_0)$ *defined by G from w_0 consists of the words $w_0, w_1, w_2, \ldots$, where for $i \geq 0$,*

$$w_{i+1} = \begin{cases} p(w_i), & \text{if } p(w_i) \in PUR \\ h_W(p(w_i)), & \text{if } p(w_i) \in PYR. \end{cases}$$

The transition $w_i \Longrightarrow_G w_{i+1}$ is also called the derivation step of G. If $w_{i+1} = h_W(p(w_i))$, then we speak about complementation derivation step. We denote $\Longrightarrow_G^*$ the transitive and reflexive closure of $\Longrightarrow_G$ as usual.

For more details and motivation underlying the concept of Watson-Crick D0L systems see [4,5,8,9] (where a Watson-Crick D0L system differs from scheme only by adding an axiom $w_0 \in PUR$). In [4] and others, a Watson-Crick D0L system is viewed also as a DT0L system with two morphisms p and h_W, together with a regulation mechanism guiding the selection of the morphism. Contrary to DT0L system, the regulation mechanism gives rise to determinism and the system generates a unique sequence of words. Another important difference is in fact that *the length of the derivation can be determined by the triggering mechanism*, as the further example shows. This is the main principle of the below proofs showing universality of Watson-Crick D0L schemes.

2 Partial Recursive Functions

In this section we briefly resume a widely-used characterization of partial recursive functions in a machine-independent way. For more details and also for famous Kleene's theorem, claiming the existence of an universal function, we refer to [11,2].

The symbol $\mathbb{N}$ is used for the set of all nonnegative integers. We denote (partial) recursive functions the class of functions $f : \mathbb{N}^t \longrightarrow \mathbb{N}$ for some $t \geq 0$, computable by Turing machines. The word "partial" can be omitted if the domain of f is $\mathbb{N}^t$. The below notation can be easily extended to functions $\mathbb{N}^t \longrightarrow \mathbb{N}^s$, $s > 0$.

Definition 2. *The family of primitive recursive functions is the smallest family of integer-to-integer functions with the following properties:*

(i) It contains the following base functions:
0 *(nullary constant),*
$S(x) = x + 1$ *(successor function),*
$U_i^n(x_1, \ldots, x_n) = x_i$ *(projection functions), for* $1 \leq i \leq n$.

(ii) It is closed under the following operations:

- composition: *if* $h : \mathbb{N}^m \longrightarrow \mathbb{N}$, $g_1 : \mathbb{N}^n \longrightarrow \mathbb{N}$, $\ldots$, $g_m : \mathbb{N}^n \longrightarrow \mathbb{N}$ *are primitive recursive functions, then so is the function* $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ *defined as follows:*

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)). \tag{1}$$

- primitive recursion: *if* $h : \mathbb{N}^n \longrightarrow \mathbb{N}$, $g : \mathbb{N}^{n+2} \longrightarrow \mathbb{N}$ *are primitive recursive functions, then so is the function* $f : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ *defined as follows:*

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= h(x_1, \ldots, x_n), \\ f(z+1, x_1, \ldots, x_n) &= g(z, f(z, x_1, \ldots, x_n), x_1, \ldots, x_n), \ z \geq 0. \end{aligned} \tag{2}$$

Theorem 1. *The family of partial recursive functions is the smallest family of integer-to-integer functions with the following properties:*

(i) It contains the nullary constant, the successor function and the projection functions.

(ii) It is closed under the operations composition, primitive recursion and minimalization*, defined as follows:*
If $h : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ *is a partial recursive function, then so is the function* $f : \mathbb{N}^n \longrightarrow \mathbb{N}$*, where*

$$f(x_1, \ldots, x_n) = \min\{y \in \mathbb{N} \,|\, h(x_1, \ldots, x_n, y) = 0\}, \tag{3}$$

and $h(x_1, \ldots, x_n, z)$ *is defined for all integers* z, $0 \leq z \leq y$. *Otherwise,* $f(x_1, \ldots, x_n)$ *is undefined.*

3 Computation by Watson-Crick D0L Schemes

In this section we define a representation of functions suitable for computing with Watson-Crick D0L schemes. Our approach is different from D0L growth functions which are frequently used in the cited literature. It is rather similar to the representation used with Turing machine. A substantial difference is the usage of different symbols for each function argument, since the Parikh vector of a D0L sequence does not depend on the order of symbols in an axiom.

In the rest of the paper, we represent an n-tuple of nonnegative numbers by the Parikh vector of a string of symbols. Both this representation and the derivation in a Watson-Crick D0L scheme is independent on the order of symbols in a string. The following definitions utilize this fact to simplify notation.

Definition 3. *Let* $G = (\Sigma, p)$ *be a Watson-Crick D0L scheme. For* $x, y \in \Sigma^*$ *we write*

$$x \dot{\Longrightarrow}_G y \quad \textit{iff} \quad x \Longrightarrow_G y',$$

where y' *is an arbitrary symbol permutation of* y. *We denote* $\dot{\Longrightarrow}^*_G$ *the transitive and reflexive closure of* $\dot{\Longrightarrow}_G$ *as usual.*

Definition 4. *Let $f : \mathbb{N}^n \longrightarrow \mathbb{N}^m$ be a (partial) function and let $G = (\Sigma, p)$ be a Watson-Crick D0L scheme. We say that G computes f if Σ^{PUR} contains subsets $\Gamma = \{\$, A_{[1]}, \ldots, A_{[n]}\}$ and $\Delta = \{\#, Z_{[1]}, \ldots, Z_{[m]}\}$ such that the following holds:*

- $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$ *iff* $\$A_{[1]}^{x_1} \ldots A_{[n]}^{x_n} \stackrel{*}{\Longrightarrow}_G \# Z_{[1]}^{y_1} \ldots Z_{[m]}^{y_m}$,
- $p(X) = X$ *for* $X \in \Delta$,
- *either* $|w|_\Delta = 0$ *or* $w \in \Delta^*$ *for each* $w \in S(G, \$A_1^{x_1} \ldots A_n^{x_n})$, *i.e. no symbol from* Δ *appears in the derivation sequence until the output string is derived.*

The sets Γ and Δ are called the *input set* and the *output set,* respectively. We can assume without loss of generality that different schemes can have disjoint alphabets, simply by renaming the symbols of Γ and Δ.

It immediately follows from the above definition that a scheme computing a partial function $f : \mathbb{N}^n \longrightarrow \mathbb{N}^m$ never produces any output symbol if $f(x_1, \ldots, x_n)$ is undefined for a certain n-tuple $x_1, \ldots, x_n$. It also follows that the result of computation is independent on order of symbols in the input string.

Example 1. The following Watson-Crick D0L scheme $G = (\Sigma, p)$ computes the function $f(n) = \lceil \log_3 n \rceil$, where $\lceil x \rceil$ is the ceiling of x.

$$\Sigma = \{\$, A_{[1]}, B, C, D, S, Z_{[1]}, \#, \overline{\$}, \overline{A_{[1]}}, \overline{B}, \overline{C}, \overline{D}, \overline{S}, \overline{Z_{[1]}}, \overline{\#}\},$$

$$\begin{array}{llll} p(\$) = \overline{S}B & p(S) = \# & p(B) = \lambda & p(\overline{S}) = \overline{S}C\overline{D} \\ p(\overline{C}) = \lambda & p(\overline{A_{[1]}}) = \lambda & p(\overline{B}) = \overline{B}B\overline{B} & p(D) = Z_{[1]} \end{array}$$

and $p(X) = X$ for all other $X \in \Sigma$. At the first step, the symbols $\overline{S}$ and $\overline{B}$ are generated, while $A_{[1]}$'s representing the value of n remain unchanged. Then at each step the number of $\overline{B}$'s is triplicated, and a pair $C\overline{D}$ is added. When the number of $\overline{B}$'s reaches the number of $A_{[1]}$'s, the complementary transition occurs, resulting in rewritting D to the output symbol $Z_{[1]}$. For example,

$$\begin{aligned} \$A_{[1]}^9 \Longrightarrow \overline{S}BA_{[1]}^9 \Longrightarrow \overline{S}C\overline{D}B^3A_{[1]}^9 \Longrightarrow \overline{S}(C\overline{D})^2\overline{B}^3A_{[1]}^9 \Longrightarrow \\ \Longrightarrow S(\overline{C}D)^3B^9\overline{A_{[1]}}^9 \Longrightarrow \#Z_{[1]}^3 \end{aligned}$$

Notice again that contrary to DT0L system, the number of output symbols is nonlinear with respect to the number of input symbols, which is allowed by the complementarity triggering mechanism.

4 Useful Properties of Watson-Crick D0L Schemes

In this section we study ability of Watson-Crick D0L schemes to compose more complex functions from simpler ones. It is shown that certain operations over functions, even simpler than those in Section 2, can be realized by Watson-Crick D0L schemes. These results are of key importance for the next section.

Definition 5. *Let $G = (\Sigma, p)$ be a Watson-Crick D0L scheme with an input set Γ and an output set Δ. G is said to be* k-lazy *if for each axiom $w_0 \in \Gamma^*$ and $q \geq 0$, $p(w_q) \in PYR$ implies $p(w_{q+i}) \in PUR$ and $w_{q+i} \notin \Delta^*$, $1 \leq i \leq k$.*

Informally, each complementation step is followed by at least k non-complementation steps during which G does not produce an output. This property will turn out important for merging several schemes into one, as following lemmata do. It follows from the above definition that if G never performs a complementation step during a derivation of $S(G, w_0)$, then G is k-lazy for any $k \geq 0$.

Definition 6. *A (partial) function f is said to be* W_k-computable*, if there is a k-lazy Watson-Crick D0L scheme computing f. f is called to be* W-computable*, if it is W_k computable for some $k \geq 0$.*

Lemma 1. *Let $f : \mathbb{N}^n \longrightarrow \mathbb{N}^m$ be a W_i-computable function, $i \geq 0$. Let $\ell \geq n$, $k \leq m$, let P and Q be arbitrary permutations over the sets $\{1, \ldots, \ell\}$ and $\{1, \ldots, m\}$, respectively. Then the following function $g : \mathbb{N}^\ell \longrightarrow \mathbb{N}^k$ is W_i-computable:*

$$g(x_{P(1)}, \ldots, x_{P(\ell)}) = (y_{Q(1)}, \ldots, y_{Q(k)}) \text{ iff } f(x_1, \ldots, x_n) = (y_1, \ldots, y_m).$$

Proof. Denote $G = (\Sigma, p)$ a scheme computing f, with the input set $\Gamma = \{\$, A_{[1]}, \ldots, A_{[n]}\}$ and the output set $\Delta = \{\#, Z_{[1]}, \ldots, Z_{[m]}\}$. Consider the scheme $G' = (\Sigma', p')$ with the input set $\Gamma' = \{\$', A'_{[1]}, \ldots, A'_{[\ell]}\}$ and the output set $\Delta' = \{\#', Z'_{[1]}, \ldots, Z'_{[k]}\}$, where $\Sigma' = \Sigma \cup \Gamma' \cup \Delta' \cup h_W(\Gamma' \cup \Delta')$,

$$p'(\$') = \$ \qquad p'(A'_{[P(j)]}) = \begin{cases} A_{[j]} & 1 \leq j \leq n, \\ \lambda & n < j \leq \ell, \end{cases}$$
$$p'(\#) = \#' \qquad p'(Z_{[j]}) = \begin{cases} Z'_{[Q(j)]} & 1 \leq j \leq k, \\ \lambda & k < j \leq m, \end{cases}$$

$p'(X) = p(X)$ for $X \in (\Sigma - \Delta)$ and $p'(X) = X$ for all other $X \in \Sigma'$. Then obviously G' computes g.

Lemma 2. *Let functions $f : \mathbb{N}^n \longrightarrow \mathbb{N}^m$ and $g : \mathbb{N}^m \longrightarrow \mathbb{N}^\ell$ be W_i and W_j-computable, respectively. Then the function $h : \mathbb{N}^n \longrightarrow \mathbb{N}^\ell$, $h(x_1, \ldots, x_n) = g \circ f(x_1, \ldots, x_n)$, is W_k-computable, where $k = \min(i, j)$ and $g \circ f$ is the composition of the functions f and g.*

Proof. We need only to rewrite output symbols of the scheme computing f to input symbols of the scheme computing g. We refer to [12] for details.

The following two lemmata show that it is possible to "call" one function from another via Watson-Crick D0L schemes, having saved an actual state of computation of the calling function.

Lemma 3. *Let $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ be a function computed by a k-lazy Watson-Crick D0L scheme F, $k \geq 1$, with an input set $\Gamma = \{\$, A_{[1]}, \ldots, A_{[n]}\}$ and an output set $\Delta = \{\#, Z\}$. Then there is a k-lazy Watson-Crick D0L scheme $G_1 = (\Sigma_1, p_1)$ such that $y = f(x_1, \ldots, x_n)$ iff*

$$\$' A'^{x_1}_{[1]} \ldots A'^{x_n}_{[n]} \stackrel{\cdot}{\Longrightarrow}^{+}_{G_1} \#Z^y (B_{[1]}\overline{C_{[1]}})^{x_1} \ldots (B_{[n]}\overline{C_{[n]}})^{x_n},$$

where $\$', A'_{[1]}, \ldots, A'_{[n]}, B_{[1]}, \ldots, B_{[n]}, \overline{C_{[1]}}, \ldots, \overline{C_{[n]}} \in \Sigma_1$.

Proof. Denote $F = (\Sigma, p)$, and let

$$\Sigma_1^{\mathrm{PUR}} = \Sigma^{\mathrm{PUR}} \cup \{\$', A'_{[j]}, B_{[j]}, C_{[j]}, C_{[j]i} \mid 1 \le j \le n,\ 1 \le i \le k\},$$

the sets on the the right hand side being disjoint. Let $\Sigma_1 = \Sigma_1^{\mathrm{PUR}} \cup h_W(\Sigma_1^{\mathrm{PUR}})$.

Let $p_1(X) = p(X)$ for each $X \in \Sigma$. Consider an arbitrary input string $w_0 = \$A_{[1]}^{x_1} \dots A_{[n]}^{x_n}$ and denote $w_0, w_1, w_2, \dots$ members of the sequence $S(G, w_0)$. Below we define elements of p_1 and simultaneously show their effect on the derived string.

Permutation of derived strings	Elements of p_1
$\$' A'^{x_1}_{[1]} \dots A'^{x_n}_{[n]}$	
$\Downarrow \cdot$	$p_1(\$') = \$,\ p_1(A'_{[j]}) = A_{[j]} B_{[j]} \overline{C_{[j]}}$, $1 \le j \le n$
$w_0 (B_{[1]} \overline{C_{[1]}})^{x_1} \dots (B_{[n]} \overline{C_{[n]}})^{x_n}$	
$\Downarrow \cdot$	$p_1(B_{[j]}) = B_{[j]},\ p_1(\overline{C_{[j]}}) = \overline{C_{[j]}}$, $1 \le j \le n$
$w_1 (B_{[1]} \overline{C_{[1]}})^{x_1} \dots (B_{[n]} \overline{C_{[n]}})^{x_n}$	
$\vdots$	

Denote for brevity

$$\alpha = (B_{[1]} \overline{C_{[1]}})^{x_1} \dots (B_{[n]} \overline{C_{[n]}})^{x_n}. \tag{4}$$

Consider now that $p(w_r) \in \mathrm{PYR}$ for some $r \ge 0$ and notice that $p_1(w_r \alpha) \in PYR$ *iff* $p(w_r) \in PYR$. Then G_1 performs a complementation step and its behavior will be the following:

Permutation of derived strings	Elements of p_1
$\vdots$	
$w_r \alpha$	
$\Downarrow \cdot$	$p_1(\alpha) = \alpha$, see above
$w_{r+1} h(\alpha)$	
$\Downarrow \cdot$	$p_1(\overline{B_{[j]}}) = \lambda,\ p_1(C_{[j]}) = B_{(j)} \overline{C_{[j]1}}$, $1 \le j \le n$
$w_{r+2} (B_{[1]} \overline{C_{[1]1}})^{x_1} \dots (B_{[n]} \overline{C_{[n]1}})^{x_n}$	
$\Downarrow \cdot_*$	$p_1(\overline{C_{[j]i}}) = \overline{C_{[j]i+1}}$, $1 \le j \le n,\ 1 < i \le k$
$w_{r+k+1} (B_{[1]} \overline{C_{[1]k}})^{x_1} \dots (B_{[n]} \overline{C_{[n]k}})^{x_n}$	

(5)

$$\Downarrow . \qquad \boxed{\begin{array}{c} p_1(\overline{C_{[j]k}}) = \overline{C_{[j]}} \\ 1 \le j \le n \end{array}}$$

$$\begin{array}{ll} w_{r+k+2}\,\alpha & \text{for } p(w_{r+k+1}) \in PUR, \text{ or} \\ w_{r+k+2}\,h(\alpha) & \text{for } p(w_{r+k+1}) \in PYR. \end{array}$$

These strings contain the same symbols as the previously derived $w_r\alpha$ or $w_{r+1}h(\alpha)$, respectively, and the derivation will continue as above. The above derivation holds since F is k-lazy by assumption and hence $p(w_{r+i}) \in PUR$, $1 \le i \le k$. All yet undefined elements of p_1 adopt the form $p_1(X) = X$, $X \in \Sigma_1$.

Now consider that in a q-th step, $q \ge 1$, the string $w_q\beta$ was derived by G_1, where $w_q = \#Z^y \in \Delta^*$, for some $\beta \in \Sigma_1^*$. Then either $w_q\beta$ adopts the form (5), if complementation occurred in $(q-k)$-th step, or $\beta = \alpha$, if complementation did not occur in the last k steps since F is k-lazy. In both cases $\#Z^y\beta \Longrightarrow_{G_1} \#Z^y\alpha$ since $\#Z^y\beta \in PUR$, and hence a permutation of the string $\#Z^y\alpha$ is produced by G_1 as the lemma claimed.

Lemma 4. *Let $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ be a W_k computable function, $k \ge 1$. Then the function $g : \mathbb{N}^n \longrightarrow \mathbb{N}^{n+1}$, $g(x_1, \ldots, x_n) = (x_1, \ldots, x_n, f(x_1, \ldots, x_n))$, is W_{k-1}-computable.*

Proof. Proof of Lemma 3 showed that there is a scheme G_1 such that $y = f(x_1, \ldots, x_n)$ *iff* $\$' A'^{x_1}_{[1]} \ldots A'^{x_n}_{[n]} \stackrel{\cdot}{\Longrightarrow}^{+}_{G_1} \#Z^y\beta$, where $\#, Z$ did not appear in previous members of the sequence, and moreover

$$p_1(\#Z^y\beta) = \#Z^y\alpha = \#Z^y(B_{[1]}\overline{C_{[1]}})^{x_1} \ldots (B_{[n]}\overline{C_{[n]}})^{x_n},$$

with α, β as in (4), (5). Nevertheless, the symbols $B_{[j]}$, $\overline{C_{[j]}}$, $1 \le j \le n$, could appear in previous members of the sequence. Now it remains to construct a scheme G_2 which rewrites all the symbols of $\#Z^y\beta$ to its output symbols which did not appear in the sequence yet.

Let $G_2 = (\Sigma_2, p_2)$, denote Γ_2 and Δ_2 the input and the output set of G_2, respectively. Let $\Gamma_2 = \Gamma_1$, $\Delta_2 = \{\#', Z'_{[1]}, \ldots, Z'_{[n+1]}\}$. Let

$$\Sigma_2^{\mathrm{PUR}} = \Sigma_1^{\mathrm{PUR}} \cup \Delta_2 \cup \{S_i, D_i \mid 0 \le i \le 2k-1\} \cup \{C_{[j]i} \mid k < i \le 2k-1,\ 1 \le j \le n\},$$

the sets on the the right hand side being mutually disjoint. Let $\Sigma_2 = \Sigma_2^{\mathrm{PUR}} \cup h_W(\Sigma_2^{\mathrm{PUR}})$.

Let $p_2(X) = p_1(X)$ for each $X \in \Sigma_1 - \Delta - \{C_{[j]k} \mid 1 \le j \le n\}$. Hence the derivation of G_2 is identical to that of G_1 until the string $\#Z^y\beta$ is produced, since the symbols $C_{[j]k}$ never appeared during this derivation.

Having the string $\#Z^y\beta$, G_2 rewrites the symbols $\#$, Z to new symbols and performs a complementation step to rewrite β to $h(\alpha)$ (see the above proof for elements of p_1). Then $k-1$ non-complementation steps follow to keep the scheme $(k-1)$-lazy.

Permutation of derived strings	Elements of p_2
$\#Z^y\beta$	
$\Downarrow \cdot$	$p_2(\#) = \overline{S_0},\ p_2(Z) = \overline{D_0}$
$S_0 D_0^y h(\alpha)$	
$\Downarrow \cdot_*$	$p_2(S_i) = S_{i+1},\ p_2(D_i) = D_{i+1}$ $0 \le i < k-1$
$S_{k-1} D_{k-1}^y (B_{[1]}\overline{C_{[1]k-1}})^{x_1} \dots (B_{[n]}\overline{C_{[n]k-1}})^{x_n}$	
$\Downarrow \cdot$	$p_2(S_{k-1}) = \overline{S_k},\ p_2(D_{k-1}) = \overline{D_k}$
$S_k D_k^y (\overline{B_{[1]}} C_{[1]k})^{x_1} \dots (\overline{B_{[n]}} C_{[n]k})^{x_n}$	

In the last step another complementation occurred, rewriting the symbols $\overline{C_{[j]k}}$ in an other way than G_1 would do. Notice that this complementation occurs only when the symbols S_{k-1}, D_{k-1} appear in the sequence, which is possible only when a computation of $y = f(x_1, \dots, x_n)$ is finished. Until then, the derivation was fully controlled by elements of p_1.

Permutation of derived strings	Elements of p_2
$\vdots$	
$\Downarrow \cdot_*$	$p_2(S_i) = S_{i+1},\ p_2(D_i) = D_{i+1}$ $p_2(C_{[j]i}) = C_{[j]i+1}$ $1 \le j \le n,\ k \le i < 2k-1$
$S_{2k-1} D_{2k-1}^y C_{[1]2k-1}^{x_1} \dots C_{[n]2k-1}^{x_n}$	
$\Downarrow \cdot$	$p_2(S_{2k-1}) = \#'$, $p_2(D_{2k-1}) = Z'_{[n+1]}$, $p_2(C_{[j]2k-1}) = Z'_{[j]},\ 1 \le j \le n$
$\#' Z'^{x_1}_{[1]} \dots Z'^{x_n}_{[n]} Z'^{y}_{[n+1]}$	

The last $k-1$ non-complementation steps was again to keep the scheme $(k-1)$-lazy. All yet undefined elements of p_2 adopt the form $p_2(X) = X$, $X \in \Sigma_2$. It follows immediately from the above description that G_2 is $(k-1)$-lazy and computes g, since the symbols from Δ_2 did not appear in the derivation sequence until the output string was produced.

Note. Consider that in the above lemma the function f is W_k-computable for any $k \ge 0$, and hence the scheme computing f never performs a complementation step. On the contrary, the resulting scheme G_2 computing g *does* perform a complementation. Hence an arbitrary but fixed lazy value k must be chosen, according to which the construction of G_2 is done, and the function g is then W_{k-1} computable.

For the next lemma we fix the following notation: let $f : \mathbb{N}^n \longrightarrow \mathbb{N}^n$ be a (partial) function. Denoting $\circ$ the composition operation, define

$$f^0(x_1, \ldots, x_n) = (x_1, \ldots, x_n),$$
$$f^k(x_1, \ldots, x_n) = f \circ f^{k-1}(x_1, \ldots, x_n),\ k > 0.$$

Lemma 5. *Let $f : \mathbb{N}^n \longrightarrow \mathbb{N}^n$, $n \geq 2$, be a W_k-computable function, $k \geq 1$. Then so is the function $g : \mathbb{N}^n \longrightarrow \mathbb{N}^n$, defined as*

$$g(x_1, \ldots, x_n) = f^i(x_1, \ldots, x_n),$$
$$i = \min\{\ell \mid f^\ell(x_1, \ldots, x_n) = (y_1, \ldots, y_n),\ y_1 \geq y_2\},$$

if such an i exists, otherwise $g(x_1, \ldots, x_n)$ is undefined.

Proof. The principle is in computation of $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and rewritting output symbols to the input ones, forming a loop. When the condition $y_1 \geq y_2$ holds, the complementation occurs, breaking the loop. We refer to [12] for details.
Notice that if no such i exists for a certain n-tuple $x_1, \ldots, x_n$, then $g(x_1, \ldots, x_n)$ is undefined and the resulting scheme computing g never produces an output symbol according to Definition 4.

5 Main Result

We show that any partial recursive function can be computed by a Watson-Crick D0L scheme. The proof is based on the fact that the base functions and operations in Theorem 1 can be realized by Watson-Crick D0L schemes. Technically, these operations can be decomposed into simpler parts, which were shown to be W-computable in the previous section.

Lemma 6. *The following functions are W_k-computable for any $k \geq 0$: (i) the nullary constant 0, (ii) the successor function $S(x)$, (iii) the projection function $U_i^n(x_1, \ldots, x_n)$.*

Proof. (i) Consider the scheme $G = (\{\$, A, \#, Z, \overline{\$}, \overline{A}, \overline{\#}, \overline{Z}\}, p)$ such that $p(\$) = \#$, $p(A) = \lambda$ and $p(X) = X$ for all other $X \in \Sigma$.
(ii) Consider the scheme $G = (\{\$, \# A, Z, \overline{\$}, \overline{\#}, \overline{A}, \overline{Z}\}, p)$ such that $p(\$) = \#Z$, $p(A) = Z$ and $p(X) = X$ for all other $X \in \Sigma$.
(iii) Consider the scheme $G = (\{\$, \# A_{[1]}, \ldots, A_{[n]}, Z, \overline{\$}, \overline{\#}, \overline{A_{[1]}}, \ldots, \overline{A_{[n]}}, \overline{Z}\}, p, \$)$ such that $p(\$) = \#$, $p(A_{[i]}) = Z$, $p(A_{[j]}) = \lambda$ for $j \neq i$ and $p(X) = X$ for all other $X \in \Sigma$.

Lemma 7. *Let $h : \mathbb{N}^m \longrightarrow \mathbb{N}$, $g_1 : \mathbb{N}^n \longrightarrow \mathbb{N}$, $\ldots$, $g_m : \mathbb{N}^n \longrightarrow \mathbb{N}$ be W_i-computable functions, $i \geq 1$. Then the function $f : \mathbb{N}^n \longrightarrow \mathbb{N}$, $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$, is W_{i-1}-computable.*

Proof. We can derive according to previous lemmata that the following functions are W_{i-1}-computable:

- $f_1 : \mathbb{N}^n \longrightarrow \mathbb{N}^{n+1}$, defined as $f_1(x_1, \ldots, x_n) = (x_1, \ldots, x_n, y_1)$, where $y_1 = g_1(x_1, \ldots, x_n)$, due to Lemma 4;
- $f_2(x_1, \ldots, x_n) = (x_1, \ldots, x_n, y_1, y_2)$, where $y_2 = g_2(x_1, \ldots, x_n)$, since
  - $g_2'(x_1, \ldots, x_n, y_1) = g_2(x_1, \ldots, x_n)$ is W_k-computable due to Lemma 1,
  - $g_2''(x_1, \ldots, x_n, y_1) = (x_1, \ldots, x_n, y_1, g_2'(x_1, \ldots, x_n, y_1))$ is W_{k-1}-computable due to Lemma 4, and
  - $f_2(x_1, \ldots, x_n) = g_2'' \circ f_1(x_1, \ldots, x_n)$ is W_{k-1}-computable due to Lemma 2
- $f_3(x_1, \ldots, x_n) = (x_1, \ldots, x_n, y_1, y_2, y_3)$, where $y_3 = g_3(x_1, \ldots, x_n)$, due to Lemmata 1, 2 and 4 as in the construction of f_2 above;

$$\vdots$$

- $f_m(x_1, \ldots, x_n) = (x_1, \ldots, x_n, y_1, \ldots, y_m)$, where $y_m = g_m(x_1, \ldots, x_n)$, analogously as in the construction of f_2 above;
- $f_{m+1}(x_1, \ldots, x_n) = (x_1, \ldots, x_n, y_1, \ldots, y_m, h(y_1, \ldots, y_m))$ as in the construction of f_2 above;
- $f(x_1, \ldots, x_n) = U_{n+m+1}^{n+m+1} \circ f_{m+1}(x_1, \ldots, x_n)$ due to Lemmata 2 and 6 (iii).

We constructed the function f using the constructions of the Lemmata 1, 2, 4 and 6 in a straightforward manner, hence f is W_{i-1}-computable function.

Lemma 8. *Let $h : \mathbb{N}^n \longrightarrow \mathbb{N}$, $g : \mathbb{N}^{n+2} \longrightarrow \mathbb{N}$ be W_i-computable functions, $i \geq 1$. Then the function $f : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ defined as in (2) is W_{i-1}-computable.*

Proof. The following functions are W_{i-1}-computable:

- $S'(y, z, w, x_1, \ldots, x_n) = S(y)$ due to Lemmata 1 and 6 (ii);
- $f_1 : \mathbb{N}^{n+3} \longrightarrow \mathbb{N}^{n+4}$, def. as $f_1(y, z, w, x_1, \ldots, x_n) = (y, z, w, x_1, \ldots, x_n, y_1)$, where $y_1 = S'(y, z, w, x_1, \ldots, x_n)$, due to Lemma calling2;
- $f_2(y, z, w, x_1, \ldots, x_n) = (y, z, w, x_1, \ldots, x_n, y_1, y_2)$, where $y_2 = g(y, w, x_1, \ldots, x_n)$, analogously to the construction of f_2 in the above proof;
- $f_3(y, z, w, x_1, \ldots, x_n) = (y_1, z, y_2, x_1, \ldots, x_n)$ due to Lemma 1;
- $f_4(y, z, w, x_1, \ldots, x_n) = f_3^k(y, z, w, x_1, \ldots, x_n)$, where $k \geq 0$ is minimal such that $y \geq z$, due to Lemma 5.

Consider $y = 0$, then the calculation of $f_4(y, z, w, x_1, \ldots, x_n)$ will proceed as follows: $w := g(y, w, x_1, \ldots, x_n)$, $y := S(y)$, repeatedly while $y < z$, i.e. z-times. If we at the beginning assume $w = h(x_1, \ldots, x_n)$, then value of w is computed exactly according to (2) in Definition 2 of the primitive recursion.

Notice that if g and h are defined for all values of arguments, then so is f_4 since the computation according to Lemma 5 finishes after z cycles. It remains to show that the following functions are W_{i-1}-computable, establishing w as the value computed by the function f :

- $h_1(z, x_1, \ldots, x_n) = h(x_1, \ldots, x_n)$ due to Lemma 1;

- $h_2(z, x_1, \ldots, x_n) = (z, x_1, \ldots, x_n, h_1(z, x_1, \ldots, x_n))$ due to Lemma 4;
- $h_3(z, x_1, \ldots, x_n) = (z, x_1, \ldots, x_n, h_1(z, x_1, \ldots, x_n), 0)$ analogously to the construction of f_2 in the above proof, having 0 the nullary function;
- $h_4(z, x_1, \ldots, x_n) = (0, z, h(x_1, \ldots, x_n), x_1, \ldots, x_n)$ due to Lemma 1;
- $f_5(z, x_1, \ldots, x_n) = f_4 \circ h_4(z, x_1, \ldots, x_n)$ due to Lemma 2;
- $f(z, x_1, \ldots, x_n) = U_3^{n+3} \circ f_5(z, x_1, \ldots, x_n)$ due to Lemmata 2 and 6 (iii).

Lemma 9. *Let $h : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ be a W_i-computable function, $i \geq 1$. Then the function $f : \mathbb{N}^n \longrightarrow \mathbb{N}$, $f(x_1, \ldots, x_n) = \min\{y \in \mathbb{N} \,|\, h(x_1, \ldots, x_n, y) = 0\}$ is W_{i-1}-computable.*

Proof. The following functions are W_{i-1}-computable:

- $f_1(v, w, x_1, \ldots, x_n, y, z) = (v, h(x_1, \ldots, x_n, y), x_1, \ldots, x_n, S(y), y)$, analogously to the construction of f_1 in the proof of Lemma 8;
- $f_2(v, w, x_1, \ldots, x_n, y, z) = f_1^k(v, w, x_1, \ldots, x_n, y, z)$, where $k \geq 0$ is minimal such that $v \geq w$, due to Lemma 5.

Consider that we compute f_2 with $v = y = z = 0$, $w = 1$, then its values will be calculated as follows: $w := h(x_1, \ldots, x_n, y)$, $y := S(y)$, $z := y$, repeatedly until $h(x_1, \ldots, x_n, y) = 0$. Taking as function value the value of y in the previous step (which is stored in z at that moment), we obtain exactly the value in 3 where the minimalization is defined.
Notice that if no $y \in \mathbb{N}$ exists such that $h(x_1, \ldots, x_n, y) = 0$, or $h(x_1, \ldots, x_n, z)$ is undefined for some $z < y$, then $f(x_1, \ldots, x_n)$ is undefined and the computation of f according to Lemma 5 never produces an output symbol.

It remains to show that the following functions are W_{i-1}-computable:

- $f_3(x_1, \ldots, x_n) = f_2(0, 1, x_1, \ldots, x_n, 0, 0)$, due to Lemmata 1, 2, 4 and 6 (i),(ii), analogously to the construction of $h_1, \ldots, h_4$ and f_5 in the above proof;
- $f(x_1, \ldots, x_n) = U_{n+4}^{n+4} \circ f_3(x_1, \ldots, x_n)$ due to Lemmata 2 and 6 (iii).

Theorem 2. *Any (partial) recursive function is W-computable.*

Proof. Follows from Theorem 1 and from Lemmata 6 – 9. Notice that a necessary condition for Lemmata 7 – 9 is that all the functions that are subject to these operations must be W_i-computable, $i \geq 1$. Taking into account that the result of each elementary operation is then W_{i-1} computable, we can ask what happens when the value 0 is reached.

Fortunately, when constructing any (partial) recursive function, we always start with base functions which are W_i computable for each i, and then we apply only a fixed number of the operations. This number is of course independent on the function arguments. During the first application of these operation over the base functions, a construction according to Lemma 4 is applied and an arbitrary but fixed lazy value must be chosen (see also note following Lemma 4). Hence it is enough to choose this value larger than the number of the operations further applied, so that the lazy value 0 is never reached.

Corollary 1. *The universal function is W-computable.*

6 Conclusions

We studied the properties of Watson-Crick complementation in the framework of deterministic Lindenmayer systems. We obtained an universal computational power only by enhancing the D0L iterated morphism by a complementation mechanism, which is one of the basic DNA operations. This unexpected result may be inspiring for both biological and computational research in molecular computing. There may be also further consequences of the result in the theory of formal languages and automata. Notice, however, that for more complex functions the number of necessary symbols in the resulting Watson-Crick D0L schemes can be enormously large.

Acknowledgements

I am grateful to Arto Salomaa for inspiration, useful remarks and discussions about this paper, to Erzsebet Csuhaj-Varjú and Rudi Freund for invitation to their institutes, and to Alica Kelemenová and Jozef Kelemen for continuous support. Research was supported by the Grant Agency of Czech Republic, grants No. 201/98/P047 and 201/99/1086.

References

1. J. Dassow, G. Păun, *Regulated Rewriting in Formal Language Theory* (Springer-Verlag, Berlin, 1989).
2. J. Gruska, *Foundations of Computing* (International Thomson Computer Press, London, 1997).
3. S. Kleene, General recursive functions on natural numbers, *Mathematische Annalen* **112** (1936) 727–742.
4. V. Mihalache, A. Salomaa, Language-Theoretic Aspects of DNA Complementarity, *Theoretical Computer Science* **250** (2001) 163-178.
5. V. Mihalache, A. Salomaa, Lindenmayer and DNA: Watson-Crick D0L systems. *EATCS Bulletin* **62** (1997) 160–175.
6. G. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms* (Springer-Verlag, Berlin, 1998).
7. G. Rozenberg, A. Salomaa, *The Mathematical Theory of L systems* (Academic Press, New York, 1980).
8. A. Salomaa, Turing, Watson-Crick and Lindenmayer. Aspects of DNA complementarity, in C.S. Calude, J. Casti, M.J. Dinneen, eds., *Unconventional Models of Computation* (Springer-Verlag, Berlin, 1998) 94–107.
9. A. Salomaa, Watson-Crick walks and roads in D0L graphs, *Acta Cybernetica* **14** (1999) 179–192.
10. A. Salomaa, G. Rozenberg, eds., *Handbook of Formal Languages* (Springer-Verlag, Berlin, 1997).
11. C. H. Smith, *A Recursive Introduction to the Theory of Computation* (Springer-Verlag, Berlin, 1994).
12. P. Sosík, *Universal Computation with Watson-Crick D0L Systems,* Silesian University, Institute of Computer Science tech. report 2000/1.

Author Index

Baiocchi C. 1
Ben-Hur A. 11
Blondel V.D. 165

Cassaigne J. 165
Caucal D. 177
Ciobanu G. 190

Fernau H. 202
Ferretti C. 153
Freund R. 214
Frisco P. 226

Gruska J. 25

Imai H. 25
Imai K. 240
Iwamoto C. 240

Karhumäki J. 69
Krithivasan K. 276
Kutrib M. 252

La Torre S. 264

Martín-Vide C. 82
Mauri G. 153
Morita K. 102, 240
Mutyam M. 276

Napoli M. 264
Naughton T.J. 288
Nichitiu C. 165

Parente M. 264
Păun Gh. 82, 214

Rotaru M. 190

Sadowski Z. 300
Sénizergues G. 114
Siegelmann H.T. 11
Sosík P. 308

Tateishi K. 240

Woods D. 288

Zakharov V.A. 133
Zandron C. 153