Chengde Mao
Takashi Yokomori (Eds.)

LNCS 4287

DNA Computing

12th International Meeting on DNA Computing, DNA12
Seoul, Korea, June 2006
Revised Selected Papers

Springer

Chengde Mao Takashi Yokomori (Eds.)

DNA Computing

12th International Meeting on DNA Computing, DNA12
Seoul, Korea, June 5-9, 2006
Revised Selected Papers

江苏工业学院图书馆
藏书章

Springer

Volume Editors

Chengde Mao
Purdue University
Department of Chemistry
560 Oval Drive, West Lafayette, IN 47907-2084, USA
E-mail: mao@purdue.edu

Takashi Yokomori
Waseda University
Faculty of Education and Integrated Arts and Sciences
Department of Mathematics
1-6-1 Nishiwaseda, Shinjuku-ku, Tokyo 169-8050, Japan
E-mail: yokomori@waseda.jp

Library of Congress Control Number: 2006938335

CR Subject Classification (1998): F.1, F.2.2, I.2.9, J.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-49024-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-49024-1 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2006
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11925903 06/3142 5 4 3 2 1 0

Preface

This volume is based on papers presented at the 12th International Meeting on DNA Computing (DNA12), which was held during June 5–9, 2006 at Seoul National University, Seoul, South Korea. DNA computing is an interdisciplinary field across computer science, mathematics, molecular biology, chemistry, physics, and nanotechnology. The central theme is to develop novel computing paradigms based on DNA. The annual meeting on DNA computing provides a major forum for scientists to present and discuss their latest results and promotes interactions between experimentalists and theoreticians.

The DNA12 Program Committee received 72 submissions and the current volume contains a selection of 34 papers from the preliminary proceedings. All selected papers were significantly revised by the authors according to the discussion during the meeting. It is our intention to cover all major areas in DNA computing, including demonstrations of biomolecular computing, theoretical models of biomolecular computing, biomolecular algorithms, in vitro and in vivo computational processes, analysis and theoretical models of laboratory techniques, biotechnological and other applications of DNA computing, DNA nanostructures, DNA nanodevices, DNA error evaluation and correction, in vitro evolution, molecular design, self-assembly systems, nucleic acid chemistry, and simulation tools. However, some papers on experimental works are not included because the authors would like to publish their works in more traditional journals.

We have organized the current volume by classifying 34 papers into 8 categories whose topical section headings (and breakdowns) are: Molecular and Membrane Computing Models (6), Complexity Analysis (3), Sequence and Tile Designs and Their Properties (5), DNA Tile Self-Assembly Models (4), Simulator and Software for DNA Computing (4), DNA Computing Algorithms and New Applications (4), Novel Experimental Approaches (3), and Experimental Solutions (5).

The editors would like to thank all participants, referees, the Program Committee, the Organization Committee, all assistants, and all sponsors for making this conference and this volume possible.

September 2006

Chengde Mao
Takashi Yokomori

Organization

Steering Committee

Lila Kari (Chair)	University of Western Ontario
Loenard Adleman (Honorary member)	University of Southern California
Anne Condon	University of British Columbia
Masami Hagiya	University of Tokyo
Natasha Jonoska	University of South Florida
Chengde Mao	Purdue University
Giancarlo Mauri	University of Milan, Bicocca
Satoshi Murata	Tokyo Institute of Technology
Gheorghe Paun	Romanian Academy and Sevilla University
John H. Reif	Duke University
Grzegorz Rozenberg	University of Leiden
Nadrian Seeman	New York University
Andrew Turberfield	University of Oxford
Erik Winfree	California Institute of Technology

Program Committee

Yaakov Benenson	Harvard University
Junghuei Chen	University of Delaware
Anne Condon	University of British Columbia
Robert M. Corn	University of California, Irvine
Max H. Garzon	University of Memphis
Hendrik Jan Hoogeboom	Leiden University
Natasha Jonoska	University of South Florida
Lila Kari	University of Western Ontario
Thomas H. LaBean	Duke University
Chengde Mao (Co-chair)	Purdue University
Satoshi Murata	Tokyo Institute of Technology
Gheorghe Paun	Romanian Academy and Sevilla University
Nadrian C. Seeman	New York University
Dipankar Sen	Simon Fraser University
William M. Shih	Harvard Medical School
Friedrich C. Simmel	University of Munich
Lloyd M. Smith	University of Wisconsin, Madison
Petr Sosik	Opava University
Milan N. Stojanovic	Columbia University
Erik Winfree	California Institute of Technology

此为试读,需要完整PDF请访问: www.ertongbook.com

Masahito Yamamoto	Hokkaido University
Hao Yan	Arizona State University
Takashi Yokomori (Co-chair)	Waseda University
Bernard Yurke	Lucent Technologies
Claudio Zandron	University of Milan, Bicocca
Byoung-Tak Zhang	Seoul National University

External Reviewers

A. Alhazov
F. Bernardini
D. Besozzi
R. Brijder
L. Cienciala
P. Dario
C. Ferretti
R. Freund
T. Fujii
A. Kameda
S. Kashiwamura
M. Hagiya
A. Kelemenov
S. Kobayashi
K. Komiya
H. Ono
A. Leporati
U. Majumder
A. Paton
K. Sadakane
J. Schaeffer
A. Suyama
S. Sahu
F. Tanaka
D. Tulpan
M. Yamamura
M. Yamashita
P. Yin

Sponsoring Institutions

Center for Bioinformation Technology (CBIT) of Seoul National University
CT & D, Inc.
Digital Genomics, Inc.
GenoProt, Inc.
Japan Ministry of Education, Sports and Culture and Sciences (MEXT)
Korea Information Science Society (SIG Bioinformation Tech.)
Ministry of Industry, Commerce and Energy of Korea
Ministry of Science and Technology of Korea (KOSEF/NRL Program)
Nano Systems Institute (NSI) of Seoul National University
Research Foundation of Seoul National University
Super Intelligence Technology Center (SITC) of Inha University
US Air Force Research Laboratory (AFRL/IFTC)

Table of Contents

Molecular and Membrane Computing Models

Complexity Analysis

Sequence and Tile Designs and Their Properties

DNA Tile Self-assembly Models

Simulator and Software for DNA Computing

DNA Computing Algorithms and New Applications

Novel Experimental Approaches

Experimental Solutions

Computing with Spiking Neural P Systems: Traces and Small Universal Systems

Mihai Ionescu[1], Andrei Păun[2],
Gheorghe Păun[3,4], and Mario J. Pérez-Jiménez[4]

[1] Research Group on Mathematical Linguistics
Universitat Rovira i Virgili
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
armandmihai.ionescu@urv.net
[2] Department of Computer Science, Louisiana Tech University
Ruston, PO Box 10348, Louisiana, LA-71272 USA, and
Universidad Politécnica de Madrid – UPM, Faculdad de Informatíca
Campus de Montegancedo s/n, Boadilla del Monte
28660 Madrid, Spain
apaun@latech.edu
[3] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucharest, Romania
george.paun@imar.ro
[4] Department of Computer Science and AI, University of Sevilla -
Avda Reina Mercedes s/n, 41012 Sevilla, Spain
gpaun@us.es, marper@us.es

Abstract. Recently, the idea of spiking neurons and thus of computing by spiking was incorporated into membrane computing, and so-called spiking neural P systems (abbreviated SN P systems) were introduced. Very shortly, in these systems neurons linked by synapses communicate by exchanging identical signals (spikes), with the information encoded in the distance between consecutive spikes. Several ways of using such devices for computing were considered in a series of papers, with universality results obtained in the case of computing numbers, both in the generating and the accepting mode; generating, accepting, or processing strings or infinite sequences was also proved to be of interest.

In the present paper, after a short survey of central notions and results related to spiking neural P systems (including the case when SN P systems are used as string generators), we contribute to this area with two (types of) results: (i) we produce small universal spiking neural P systems (84 neurons are sufficient in the basic definition, but this number is decreased to 49 neurons if a slight generalization of spiking rules is adopted), and (ii) we investigate the possibility of generating a language by following the trace of a designated spike in its way through the neurons.

1 Introduction

Spiking neural P systems (in short, SN P systems) were introduced in [6], with the motivation coming from two directions: the attempt of membrane computing

C. Mao and T. Yokomori (Eds.): DNA12, LNCS 4287, pp. 1–16, 2006.
© Springer-Verlag Berlin Heidelberg 2006

此为试读,需要完整PDF请访问: www.ertongbook.com

to pass from cell-like architectures to tissue-like or neural-like architectures (see [15], [12]), and the intriguing possibility of encoding information in the duration of events, or in the interval of time elapsed between events, as vividly investigated in recent research *in neural computing* (of "third generation") [8], [9].

This double challenge led to a class of P systems based on the following simple ideas: let us use only one object, the symbol denoting a *spike*, and one-membrane cells (called *neurons*) which can hold any number of spikes; each neuron fires in specified conditions (after collecting a specified number of spikes) and then sends one spike along its axon; this spike passes to all neurons connected by a *synapse* to the spiking neuron (hence it is replicated into as many copies as many target neurons exist); between the moment when a neuron fires and the moment when it spikes, each neuron needs a time interval, and this time interval is the essential ingredient of the system functioning (the basic information carrier – with the mentioning that also the number of spikes accumulated in each moment in the neurons provides an important information for controlling the functioning of the system); one of the neurons is considered the output one, and its spikes provide the output of the computation. The sequence of time moments when spikes are sent out of the system is called a *spike train*. The rules for spiking take into account *all* spikes present in a neuron not only part of them, but not all spikes present in a neuron are consumed in this way; after getting fired and before sending the spike to its synapses, the neuron is idle (biology calls this the refractory period) and cannot receive spikes. There are also rules used for "forgetting" some spikes, rules which just remove a specified number of spikes from a neuron.

In the spirit of spiking neurons, as the result of a computation (not necessarily a halting one) in [6] one considers the number of steps elapsed between the first two spikes of the output neuron. Even in this restrictive framework, SN P systems turned out to be Turing complete, *able to compute all Turing computable sets of natural numbers*. This holds both in the generative mode (as sketched above, a number is computed if it represents the interval between the two consecutive spikes of the output neuron) and in the accepting mode (a number is introduced in the system in the form of the interval of time between the first two spikes entering a designated neuron, and this number is accepted if the computation halts). If a bound is imposed on the number of spikes present in any neuron during a computation, then *a characterization of semilinear sets of numbers is obtained*.

These results were extended in [13] to several other ways of associating a set of numbers with an SN P system: taking into account the interval between the first k spikes of each spike train, or all spikes, taking only alternately the intervals, or all of them, considering halting computations. Then, the spike train itself (the sequences of symbols 0, 1 describing the activity of the output neuron: we write 0 if no spike exits the system in a time unit and 1 if a spike is emitted) was considered as the result of a computation; the infinite case is investigated in [14], the finite one in [2]. A series of possibilities of handling infinite sequences of bits are discussed in [14], while morphic representations of regular and of recursively

enumerable languages are found in [2]. The results from [2] are briefly recalled in Section 5 below.

In this paper we directly continue these investigations, contributing in two natural directions. First, the above mentioned universality results (the possibility to compute all Turing computable sets of numbers) do not give an estimation on the number of neurons sufficient for obtaining the universality. Which is the size of the smallest universal "brain" (of the form of an SN P system)? This is both a natural and important (from computer science and, also, from neuro-science point of view) problem, reminding the extensive efforts paid for finding small universal Turing machines – see, e.g., [16] and the references therein.

Our answer is rather surprising/encouraging: *84 neurons ensure the universality* in the basic setup of SN P systems, as they were defined in [6], while this number is decreased to 49 if slightly more general spiking rules are used (rules with the possibility to produce not only one spike, *but also two or more spikes at the same time* – such rules are called *extended*). The proof is based on simulating a small universal register machine from [7]. (The full details for the proof of these results about small universal SN P systems will be provided elsewhere – see [11].)

Extended rules are also useful when generating strings: we associate a symbol b_i with a step when the system outputs i spikes and in this way we obtain a string over an arbitrary alphabet, not only on the binary one, as in the case of standard rules. Especially flexible is the case when we associate the empty string with a step when no spike is sent out of the system we associate (that is, b_0 is interpreted as λ). Results from [3], concerning the power of extended SN P systems as language generators, are also recalled in Section 5.

Then, another natural issue is to bring to the SN P systems area a notion introduced for symport/antiport P systems in [5]: mark a spike and follow its path through the system, recording the labels of the visited neurons until either the marking disappears or the computation halts. Because of the very restrictive way of generating strings in this way, there are simple languages which cannot be computed, but, on the other hand, there are rather complex languages which can be obtained in this framework.

Due to space restrictions, we do not give full formal details in definitions and proofs (we refer to the above mentioned papers for that); such details are or will be available in separate papers to be circulated/announced through [19].

2 Formal Language Theory Prerequisites

We assume the reader to be familiar with basic language and automata theory, e.g., from [17] and [18], so that we introduce here only some notations and notions used later in the paper.

For an alphabet V, V^* denotes the set of all finite strings of symbols from V; the empty string is denoted by λ, and the set of all nonempty strings over V is denoted by V^+. When $V = \{a\}$ is a singleton, then we write simply a^* and a^+ instead of $\{a\}^*, \{a\}^+$. If $x = a_1 a_2 \dots a_n$, $a_i \in V$, $1 \leq i \leq n$, then the mirror image of x is $mi(x) = a_n \dots a_2 a_1$.

A morphism $h : V_1^* \longrightarrow V_1^*$ such that $h(a) \in \{a, \lambda\}$ for each $a \in V_1$ is called a projection, and a morphism $h : V_1^* \longrightarrow V_2^*$ such that $h(a) \in V_2 \cup \{\lambda\}$ for each $a \in V_1$ is called a weak coding.

If $L_1, L_2 \subseteq V^*$ are two languages, the left and right quotients of L_1 with respect to L_2 are defined by $L_2 \backslash L_1 = \{w \in V^* \mid xw \in L_1 \text{ for some } x \in L_2\}$, and respectively $L_1 / L_2 = \{w \in V^* \mid wx \in L_1 \text{ for some } x \in L_2\}$. When the language L_2 is a singleton, these operations are called left and right derivatives, and denoted by $\partial_x^l(L) = \{x\} \backslash L$ and $\partial_x^r(L) = L / \{x\}$, respectively.

A Chomsky grammar is given in the form $G = (N, T, S, P)$, where N is the nonterminal alphabet, T is the terminal alphabet, $S \in N$ is the axiom, and P is the finite set of rules. For regular grammars, the rules are of the form $A \to aB, A \to a$, for some $A, B \in N, a \in T$.

We denote by FIN, REG, CF, CS, RE the families of finite, regular, context-free, context-sensitive, and recursively enumerable languages; by MAT we denote the family of languages generated by matrix grammars without appearance checking. The family of Turing computable sets of numbers is denoted by NRE (these sets are length sets of RE languages, hence the notation).

Let $V = \{b_1, b_2, \ldots, b_m\}$, for some $m \geq 1$. For a string $x \in V^*$, let us denote by $val_m(x)$ the value in base $m+1$ of x (we use base $m+1$ in order to consider the symbols $b_1, \ldots, b_m$ as digits $1, 2, \ldots, m$, thus avoiding the digit 0 in the left hand of the string). We extend this notation in the natural way to sets of strings.

All universality results of the paper are based on the notion of a register machine. Such a device – in the non-deterministic version – is a construct $M = (m, H, l_0, l_h, I)$, where m is the number of registers, H is the set of instruction labels, l_0 is the start label (labeling an ADD instruction), l_h is the halt label (assigned to instruction `HALT`), and I is the set of instructions; each label from H labels only one instruction from I, thus precisely identifying it. The instructions are of the following forms:

- l_i : (`ADD`$(r), l_j, l_k$) (add 1 to register r and then go to one of the instructions with labels l_j, l_k non-deterministically chosen),
- l_i : (`SUB`$(r), l_j, l_k$) (if register r is non-empty, then subtract 1 from it and go to the instruction with label l_j, otherwise go to the instruction with label l_k),
- l_h : `HALT` (the halt instruction).

A register machine M generates a set $N(M)$ of numbers in the following way: we start with all registers empty (i.e., storing the number zero), we apply the instruction with label l_0 and we continue to apply instructions as indicated by the labels (and made possible by the contents of registers); if we reach the halt instruction, then the number n present in register 1 at that time is said to be generated by M. (Without loss of generality we may assume that in the halting configuration all other registers are empty; also, we may assume that register 1 is never subject of SUB instructions, but only of ADD instructions.) It is known (see, e.g., [10]) that register machines generate all sets of numbers which are Turing computable.

A register machine can also be used as a number accepting device: we introduce a number n in some register r_0, we start working with the instruction with label l_0, and if the machine eventually halts, then n is accepted (we may also assume that all registers are empty in the halting configuration). Again, accepting register machines characterize NRE.

Furthermore, register machines can compute all Turing computable functions: we introduce the numbers $n_1, \ldots, n_k$ in some specified registers $r_1, \ldots, r_k$, we start with the instruction with label l_0, and when we stop (with the instruction with label l_h) the value of the function is placed in another specified register, r_t, with all registers different from r_t being empty. Without loss of generality we may assume that $r_1, \ldots, r_k$ are the first k registers of M, and then the result of the computation is denoted by $M(n_1, \ldots, n_k)$.

In both the accepting and the computing case, the register machines can be *deterministic*, i.e., with the ADD instructions of the form $l_i : (\texttt{ADD}(r), l_j)$ (add 1 to register r and then go to the instruction with label l_j).

In the following sections, when comparing the power of two language generating/accepting devices the empty string λ is ignored.

3 Spiking Neural P Systems

We give here the basic definition we work with, introducing SN P systems in the form considered in the small universal SN P systems, hence computing functions (which, actually, covers both the generative and accepting cases).

A computing *spiking neural membrane system* (abbreviated SN P system), of degree $m \geq 1$, is a construct of the form

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out),$$

where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. $\sigma_1, \ldots, \sigma_m$ are *neurons*, of the form

$$\sigma_i = (n_i, R_i), 1 \leq i \leq m,$$

   where:
   a) $n_i \geq 0$ is the *initial number of spikes* contained in σ_i;
   b) R_i is a finite set of *rules* of the following two forms:
      (1) $E/a^c \rightarrow a; d$, where E is a regular expression[1] over a, $c \geq 1$, and $d \geq 0$;
      (2) $a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a; d$ of type (1) from R_i, we have $a^s \notin L(E)$;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses* between neurons);
4. $in, out \in \{1, 2, \ldots, m\}$ indicate the *input* and the *output* neurons of Π.

[1] The regular language defined by E is denoted by $L(E)$.

The rules of type (1) are *firing* (we also say *spiking*) *rules*, and they are applied as follows. If the neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \to a; d \in R_i$ can be applied. This means consuming (removing) c spikes (thus only $k-c$ remain in σ_i), the neuron is fired, and it produces a spike after d time units (as usual in membrane computing, a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized). If $d = 0$, then the spike is emitted immediately, if $d = 1$, then the spike is emitted in the next step, etc. If the rule is used in step t and $d \geq 1$, then in steps $t, t+1, t+2, \ldots, t+d-1$ the neuron is *closed* (this corresponds to the refractory period from neurobiology), so that it cannot receive new spikes (if a neuron has a synapse to a closed neuron and tries to send a spike along it, then that particular spike is lost). In the step $t+d$, the neuron spikes and becomes again open, so that it can receive spikes (which can be used starting with the step $t+d+1$).

The rules of type (2) are *forgetting* rules; they are applied as follows: if the neuron σ_i contains exactly s spikes, then the rule $a^s \to \lambda$ from R_i can be used, meaning that all s spikes are removed from σ_i.

If a rule $E/a^c \to a; d$ of type (1) has $E = a^c$, then we will write it in the following simplified form: $a^c \to a; d$. If all spiking rules are of this form, then the system is said to be *finite* (it can handle only a bounded number of spikes in each of its neurons).

In each time unit, if a neuron σ_i can use one of its rules, then a rule from R_i *must* be used. Since two firing rules, $E_1/a^{c_1} \to a; d_1$ and $E_2/a^{c_2} \to a; d_2$, can have $L(E_1) \cap L(E_2) \neq \emptyset$, it is possible that two or more rules can be applied in a neuron, and in that case, only one of them is chosen non-deterministically. Note however that, by definition, if a firing rule is applicable, then no forgetting rule is applicable, and vice versa.

Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other (the system is synchronized).

The initial configuration of the system is described by the numbers $n_1, n_2, \ldots, n_m$, of spikes present in each neuron, with all neurons being open. During the computation, a configuration is described by both the number of spikes present in each neuron and by the state of the neuron, more precisely, by the number of steps to count down until it becomes open (this number is zero if the neuron is already open).

A computation in a system as above starts in the initial configuration. In order to compute a function $f : \mathbf{N}^k \longrightarrow \mathbf{N}$, we introduce k natural numbers $n_1, \ldots, n_k$ in the system by "reading" from the environment a binary sequence $z = 0^b 10^{n_1-1} 10^{n_2-1} 1 \ldots 10^{n_k-1} 10^f$, for some $b, f \geq 0$; this means that the input neuron of Π receives a spike in each step corresponding to a digit 1 from the string z. Note that we input exactly $k+1$ spikes. The result of the computation is also encoded in the distance between two spikes: we impose to the system to output exactly two spikes and halt (sometimes after the second spike), hence producing a train spike of the form $0^{b'} 10^{r-1} 10^{f'}$, for some $b', f' \geq 0$ and with $r = f(n_1, \ldots, n_k)$.

If we use an SN P system in the generative mode, then no input neuron is considered, hence no input is taken from the environment; we start from the initial configuration and the distance between the first two spikes of the output neuron (or other numbers, see the discussion in the Introduction) is the result of the computation. Dually, we can ignore the output neuron, we input a number in the system as the distance between two spikes entering the input neuron, and if the computation halts, then the number is accepted.

We do not give here examples, because in the next section we show the four basic modules of our small universal SN P system.

4 Two Small Universal SN P Systems

In both the generating and the accepting case, SN P systems are universal, they compute the Turing computable sets of numbers. The proofs from [6], [13] are based on simulating register machines, which are known to be equivalent to Turing machines when computing (generating or accepting) sets of numbers, [10]. In [7], the register machines are used for computing functions, with the universality defined as follows. Let $(\varphi_0, \varphi_1, \ldots)$ be a fixed admissible enumeration of the set of unary partial recursive functions. A register machine M_u is said to be universal if there is a recursive function g such that for all natural numbers x, y we have $\varphi_x(y) = M_u(g(x), y)$. In [7], the input is introduced in registers 1 and 2, and the result is obtained in register 0 of the machine.

$l_0 : (\texttt{SUB}(1), l_1, l_2)$, $l_1 : (\texttt{ADD}(7), l_0)$,
$l_2 : (\texttt{ADD}(6), l_3)$, $l_3 : (\texttt{SUB}(5), l_2, l_4)$,
$l_4 : (\texttt{SUB}(6), l_5, l_3)$, $l_5 : (\texttt{ADD}(5), l_6)$,
$l_6 : (\texttt{SUB}(7), l_7, l_8)$, $l_7 : (\texttt{ADD}(1), l_4)$,
$l_8 : (\texttt{SUB}(6), l_9, l_0)$, $l_9 : (\texttt{ADD}(6), l_{10})$,
$l_{10} : (\texttt{SUB}(4), l_0, l_{11})$, $l_{11} : (\texttt{SUB}(5), l_{12}, l_{13})$,
$l_{12} : (\texttt{SUB}(5), l_{14}, l_{15})$, $l_{13} : (\texttt{SUB}(2), l_{18}, l_{19})$,
$l_{14} : (\texttt{SUB}(5), l_{16}, l_{17})$, $l_{15} : (\texttt{SUB}(3), l_{18}, l_{20})$,
$l_{16} : (\texttt{ADD}(4), l_{11})$, $l_{17} : (\texttt{ADD}(2), l_{21})$,
$l_{18} : (\texttt{SUB}(4), l_0, l_h)$, $l_{19} : (\texttt{SUB}(0), l_0, l_{18})$,
$l_{20} : (\texttt{ADD}(0), l_0)$, $l_{21} : (\texttt{ADD}(3), l_{18})$,
$l_h : \texttt{HALT}$.

Fig. 1. The universal register machine from [7]

The constructions from [6] do not provide a bound on the number of neurons, but such a bound can be found if we start from a specific universal register machine. We will use here the one with 8 registers and 23 instructions from [7] – for the reader convenience, this machine is recalled in Figure 1, in the notation and the setup introduced in the previous section.

Theorem 1. *There is a universal SN P system with 84 neurons.*

此为试读,需要完整PDF请访问: www.ertongbook.com