DNA Algorithms for Petri Net Modeling

Alfons Schuster
School of Computing and Mathematics, Faculty of Engineering, University of Ulster, Shore Road, Newtownabbey, Co. Antrim BT37 0QB, Northern Ireland
E-mail: a.schuster@ulster.ac.uk
http://www.infc.ulst.ac.uk/cgi-bin/infdb/buscard?email=a.schuster

Keywords: DNA computing, algorithms, Petri nets

Received: March 3, 2008

The paper applies, in a theoretical investigation, the DNA computing paradigm to the modeling of Petri nets. A run-through example demonstrates the feasibility of the approach as well as its potential practical value.

Povzetek: Podani so DNA algoritmi za Petri mreže.

1 Introduction

A defining moment for DNA computing was Adleman's (1) fundamental contribution in which he demonstrated the potential of this novel computing paradigm by solving an instance of the Hamiltonian Path Problem in theoretical as well as practical terms. Since then, DNA computing has been proposed and tested in numerous areas including, finite automata (14), machine learning (12), relational database modeling (13), and, of course, solving computationally expensive problems (e.g., (6), (2), and (3)).

This paper investigates Petri nets as a novel DNA computing application area. The paper provides brief introductions to Petri nets and DNA computing and demonstrates via an example algorithm how the DNA computing paradigm can be successfully applied for Petri net modeling. It is necessary to mention that the the paper does not include simulations of the work on a silicon computer or practical, experimental work involving real-life DNA material. Rather, the paper is of theoretical value only and largely neglects aspects of practical realizations of the proposed work (e.g., error rates). In a sense, the algorithm presented in this work is a high-level description for a program. The program/algorithm describes a sequence of biochemical events and these events are meant to execute/run in a biochemical environment—a DNA computer. Once this sequence of events is executed correctly, which is not a trivial bioengineering task, the result is available as/in the form of DNA strings. In order to extract the outcome of the algorithm, it is necessary to readout these strings and decode their information, but this is similar to reading out a sequence in a human genome (e.g., identifying a protein-encoding gene). Perhaps, one could think of the following analogy. It is possible to add and subtract two numbers with an electronic calculator, but the same thing can be done with an abacus (the abacus made of wood). Both procedures produce the same result but use entirely different machines and very different algorithms. Similarly, a DNA computer operates in a biochemical environment, executes real biochemical events, and uses real (usually synthetically modified) DNA.

In the remainder, Section 2 provides a brief introduction to Petri nets, their design, and working. Section 3 starts with a summary on DNA computing and then describes a DNA algorithm for a Petri net example the paper uses as a run-through vehicle to explain the presented work. Section 4 provides a discussion and Section 5 ends the paper with a summary.

2 Petri nets

Petri nets are the brainchild of Carl Adam Petri (9). Since their conception, Petri nets are a very lively field where findings in theoretical and applied work are continually added to the field. A summary may describe Petri nets as a formal, graphical, executable technique for the specification and analysis of concurrent, discrete-event dynamic systems (e.g., see http://www.petrinets.info/). Over time, the field attracted a lot of interest not only in the computing community but in a diverse spectrum of application areas including software & hardware (the complete software lifecycle from analysis, specification, design, modeling, simulation, and testing; e.g., (11) and (5)), complex systems (16), particle interaction in atomic physics (10), as well as model validation of biological pathways (4), for example. This section introduces Petri nets via a simple example network. The paper uses this example network as a run-through vehicle in forthcoming sections. For a start, Figure 1 illustrates the main Petri net components (place, active place, transition, directed arc, and token).

In order to build a network, the components in Figure 1 are combined in a systematic way by a set of relatively straightforward rules. Note that some of these definitions are adopted from (8), which is one of several excellent books available on Petri nets. The main design rules are:

Figure 1: Main Petri net components.

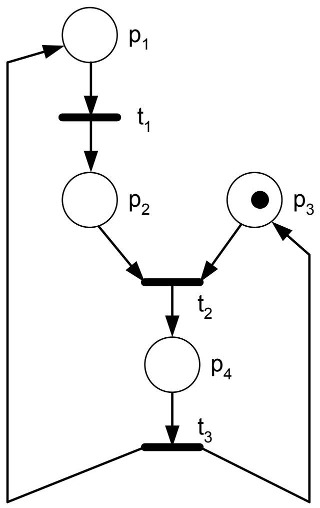


Figure 2: A Petri net with four places and three transitions.

- An arc always connects a place to a transition (in either direction).
- An arc never connects a place directly to another place nor a transition directly to another transition.
- Each place and each transition should have at least one incoming and at least one outgoing arc.
- There is no upper limit to the number of arcs that can connect to a place or a transition.

The Petri net in Figure 2, for example, is constructed by these rules. The network has four places (p_1, p_2, p_3, and p_4) and three transitions (t_1, t_2, and t_3). Directed arcs connect places and transitions, and place p_3 is an active place with one token in it. The main rules for Petri net tokens, and Petri net operations in general, are equally simple:

- Tokens are used to indicate which places are "active" (see Figure 1 and Figure 2). An active place may contain more than one token.
- If all its incoming places are active, a transition will "fire".
- When a transition fires then (a) all its incoming places lose a token, and (b) all its outgoing places gain a token.

$$P = \{p_1, p_2, p_3, p_4\}$$
$$T = \{t_1, t_2, t_3\}$$

$$I(t_1) = \{p_1\}$$
$$I(t_2) = \{p_2, p_3\}$$
$$I(t_3) = \{p_4\}$$

$$O(t_1) = \{p_2\}$$
$$O(t_2) = \{p_4\}$$
$$O(t_3) = \{p_1, p_3\}$$

Figure 3: Petri net structure $C = \{P, T, I, O\}$ for the Petri net in Figure 2.

Although Petri nets and their operations are relatively easy to understand via graphical illustrations, it is necessary mentioning that the field rests on a rigorous mathematical underpinning (e.g., see (8)). In formal terms, a Petri net is composed of four parts:

- A set P of places.
- A set T of transitions.
- An "input" function I. The input function I is a mapping from a transition t_j to a collection of places (input places) $I(t_j)$.
- An "output" function O. The output function O is a mapping from a transition t_j to a collection of places (output places) $O(t_j)$.
- The "structure" C of a Petri net is defined by its places, transitions, input function, and output function; $C = (P, T, I, O)$.

Figure 3 uses the latter definitions for the description of the Petri net in Figure 2.

It is possible to describe a Petri net and its working entirely in a rigorous mathematical way. The paper uses a simpler approach. It keeps the mathematical notation to a minimum, and instead uses graphical illustrations for the sake of ease of understanding. The paper uses Figure 4 to demonstrate the operating behavior of the Petri net in Figure 2.

Figure 4 (a) illustrates an "unmarked" Petri net. An unmarked Petri net has no tokens assigned to any of its places. In Figure 4 (b) the Petri net is "marked" with two tokens, one token is in place p_1, and the other token in place p_3. The previous text mentioned that the dynamic behavior of a Petri net is determined by the number and distribution of tokens in the Petri net. According to these rules, transition t_1 fires, because all its incoming places (p_1) have a token. Consequently, place p_1 loses its token and place p_2 gains a token (see Figure 4 (c)). Now, transition t_2 fires, because all its incoming places (p_2, and p_3) have a token. As a result, place p_2 and place p_3 lose their token, and place p_4 receives a token (see Figure 4 (d)). Next, transition t_3 fires,

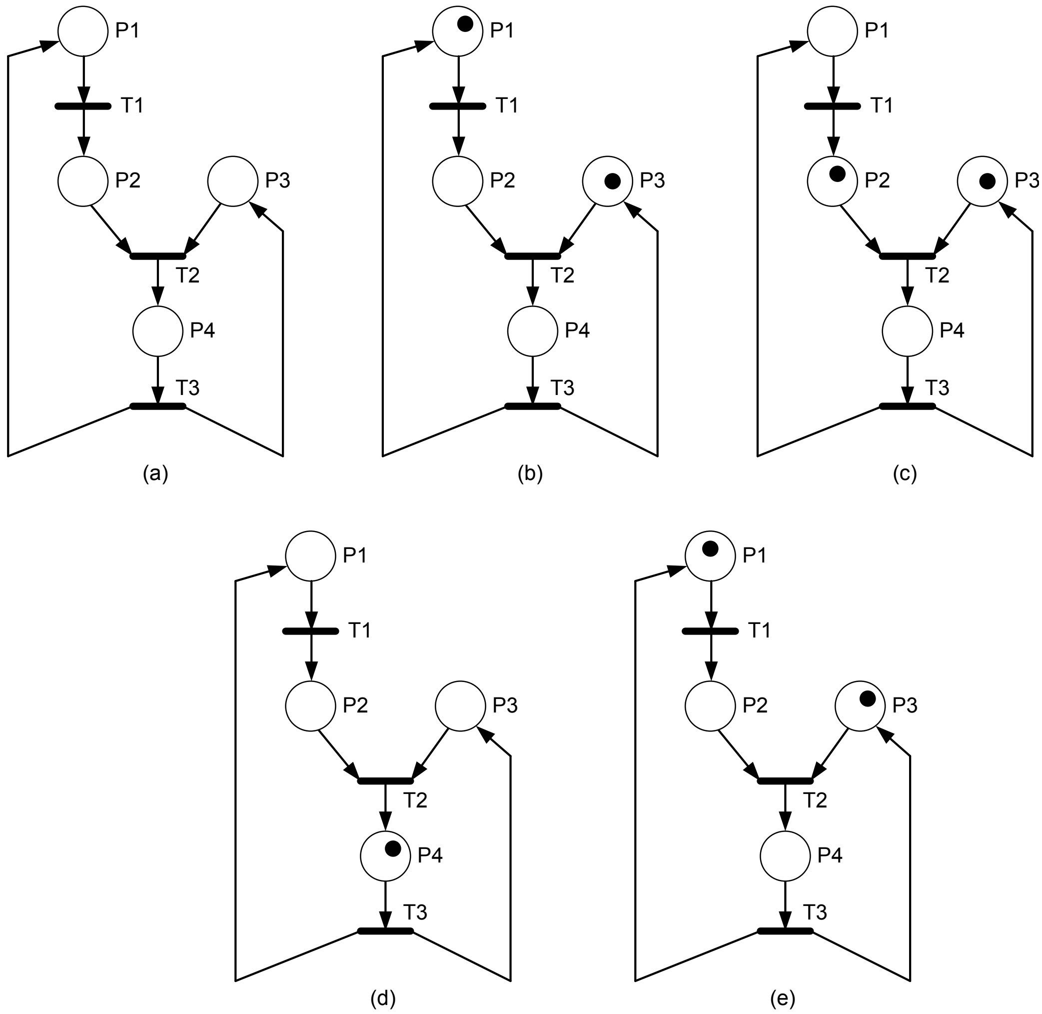


Figure 4: Execution sequence for the Petri net in Figure 2.

causing p_4 losing its token, and placing tokens into place p_1 and place p_3 (see Figure 4 (e)). A closer look reveals that Figure 4 (e) is equivalent to Figure 4 (b), and it should be clear that the example illustrates a loop.

3 DNA computing

DNA computing is a relatively young computing paradigm. Among other things, the potential of DNA computing lies in its inherent capacity for vast parallelism, the scope for high-density storage, and its intrinsic ability for potentially solving many combinatorial problems. In simple terms, DNA computing is based on the design, manipulation, and processing of nucleotides. These nucleotides are chemical compounds including a chemical base, a sugar, and a phosphate group. Four main nucleotides are distinguished, *adenine* (*A*), *guanine* (*G*), *cytosine* (*C*) and *thymine* (*T*). Nucleotides can combine or bond as "single stranded" DNA, or "double stranded" DNA. Single stranded DNA is generated through the subsequent bonding of any of the four types of nucleotides, and is often illustrated as a string of letters (e.g., TATCGGATCGGTATATCCGA). Double stranded DNA is generated from single stranded DNA and its complementary strand. This type of bonding follows "Watson-Crick Complementary", which says that base *A* only bonds with base *T*, base *G* only with base *C*, and vice versa. For example, the strand ATAGCCTAGCCATATAGGCA is the Watson-Crick Complement of the DNA strand TATCGGATCGGTATATCCGA just mentioned. The literature often illustrates a resulting double strand as two parallel strands (e.g., $\frac{TATCGGATCGGTATATCCGA}{ATAGCCTAGCCATATAGGCA}$, where the fraction line symbolizes bonding).

From a computing perspective, the field aims for the construction of DNA computers and programs that run on such a computer. Typically, the four nucleotides mentioned before provide the basis for an alphabet (e.g., Σ = {A, G, C, T}). From this alphabet a particular language (L) may be constructed. This language is used to define algorithms and computer programs. In practical terms, a DNA computer bears similarity to a biochemical machine in which biochemical events perform algorithms and execute programs by manipulating DNA strands in a series of carefully orchestrated biochemical processes. They are usually mediated by molecular entities called enzymes and include the lengthening, shortening, cutting, linking, and multiplying of DNA, for example. It is necessary to point out that these events and processes are quite challenging from a biochemical engineering perspective, but it is beyond the scope of this paper to indulge into the many challenges the field holds in this regard. There is a large body of literature available on the subject, and the interested reader is referred to one of the excellent books (7) available for the field. It may be useful however to direct a reader to some of the major contributions in the field (e.g., (15), (1), and (6)). The more imminent goal is to demonstrate the potential application of the DNA computing paradigm to the field of Petri nets.

3.1 DNA-based model for Petri nets

The goal is the design and behavioral modeling of Petri nets via DNA computing principles. The paper mentioned that the behavior of a Petri net is linked to the firing of transitions, which essentially boils down to the monitoring of activated places (i.e., places with tokens in them). For example, transition t_2 in Figure 4 (a) fires only if place p_2 and place p_3 at least hold one token each. This could be presented by a simple it-then rule: if p_2 and p_3 then t_2. The presented approach therefor has two main features:

1. It models activated places via DNA strands. For example, it is possible to represent the active place p_1 in Figure 4 (b) via the DNA strand s_1 = TATCGGATCGGTATATCCGA.
2. An algorithm describes the behavior or logic of a Petri net. This algorithm is similar to a sequence of biochemical reactions on DNA strands.

For the forthcoming sections it is necessary to introduce some of the most common operations (biochemical reactions) on DNA strands. Note that some of the following definitions are adopted from (7).

- **amplify:** Given a tube N, *amplify*(N) produces two copies of it.
- **detect:** Given a tube N, *detect*(N) returns *true* if N contains at least one DNA strand, otherwise return is *false*.
- **merge:** Given tubes N_1 and N_2, *merge*(N_1, N_2) produces a new tube N_3 that forms the union $N_1 \cup N_2$ of the two tubes.
- **separate:** Given a tube N and a DNA strand w composed of nucleotides $m \in$ {A, T, C, G}, *separate*($+N, w$) produces a new tube N_1 that consists of all strands in N which contain w as a consecutive sub-strand. Similarly, *separate*($-N, w$) produces a new tube N_1 that consists of all strands in N which do not contain w as a consecutive sub-strand.
- **lengthSeparate:** Given a tube N and an integer n, *lengthSeparate*($N, \leq n$) produces a new tube N_1 consisting of all strands in N with length less then or equal to n.

Without further ado, the paper uses Figure 5 to illustrate a DNA algorithm for the Petri net scenario in Figure 4.

The algorithm starts in line one with tube N_0. This tube is completely empty (indicated by the symbol). It is helpful to imagine that this state is equivalent to the unmarked Petri net in Figure 4 (a). The algorithm uses four boolean variables ($b1$ to $b4$ in line two) to represent the presence of any of the strands s_1 to s_4 in tube N_0. Remember that the presence of a strand is similar to an active place in the Petri net. For example, the presence of strand s_1 in tube N_0 indicates that there is a token in place p_1. The boolean variables indicate the presence of a strand by the value *true*, and its absence by the value *false*. Initially tube N_0 is empty, and so $b1 = b2 = b3 = b4 = false$ in line two. Please note that the discussion in Section 4 comments on these variables in more detail.

Line three to ten mark the Petri net. There are several possibilities for marking this particular net, and the current example uses one of them. It is possible therefore, to compare line three to ten to a function call, for instance function Mark_Petri_net(), in a computer programme where parameters are passed to the function. Line four is a simple comment. A double slash (//) always indicates a comment in the algorithm. Anyhow, line five adds strand s_1 into tube N_0, which is equivalent to adding a token into place p_1, and line six adds strand s_3 into tube N_0, which is equivalent to adding a token into place p_3. The Petri net is now in the state illustrated by Figure 4 (b). The settings of variable $b1 = true$, and $b3 = true$ in line seven and line eight reflect the presence of these strands in tube N_0. Petri nets are often models of real world systems. The appearance of a token in a place usually triggers some event in this system. This is the reason for line nine, which is a reference to some external task that my be executed by the algorithm.

Line 11 introduces the variable n. The algorithm uses this variable in the *repeat-until* loop where it defines an application specific exit criterion (line 36). The repeat-until loop extends from line 12 to line 36, and contains three *if-then* statements. Essentially, each if-then statement does two things, first, it checks the firing status of a particular transition, and second, it contains instructions for what happens when a transition fires. For example, line 14 checks for strand s_1 in tube N_0. In case the result is negative, nothing happens, and the algorithm advances to the next if-then statement. If the result is positive then place p_1 loses a token (line 15, $b1 = false$) and place p_2 gains a token (line 16, $b2 = true$). In a similar fashion, the second if-then statement monitors transition t_2, and the third if-then statement transition t_3. The comment in line 16 indicates that setting any of the boolean variables to $true$ is equal to adding a corresponding strand to tube N_1. Note that the algorithm deals with this at a later stage (lines 30 to 33).

(1) *input*(N_0) $=$
(2) $b1 = b2 = b3 = b4 = false$,
(3) **Mark Petri net begin**
(4) //For example
(5) *add*(s_1, N_0)
(6) *add*(s_3, N_0)
(7) $b1 = true$
(8) $b3 = true$
(9) Do some task.
(10) **end**
(11) $n = 0$
(12) **repeat**
(13) *input*(N_1) $=$
(14) **if** *detect*($N_0(s_1)$) **then begin**
(15) $b1 = false$
(16) $b2 = true$, //*add*(s_2, N_1) later
(17) Do some task.
(18) **end**
(19) **if** *detect*(($N_0(s_2)$ **and** $N_0(s_3)$) **then begin**
(20) $b2 = b3 = false$
(21) $b4 = true$, //*add*(s_4, N_1) later
(22) Do some task.
(23) **end**
(24) **if** *detect*($N_0(s_4)$) **then begin**
(25) $b4 = false$
(26) $b1 = true$, //*add*(s_1, N_1) later
(27) $b3 = true$, //*add*(s_3, N_1) later
(28) Do some task.
(29) **end**
(30) **if** $b1 = true$ **then** *add*(s_1, N_1)
(31) **if** $b2 = true$ **then** *add*(s_2, N_1)
(32) **if** $b3 = true$ **then** *add*(s_3, N_1)
(33) **if** $b4 = true$ **then** *add*(s_4, N_1)
(34) $N_0 = N_1$
(35) $n = n + 1$
(36) **until** (some condition regarding n is met)

Figure 5: DNA-based algorithm for the Petri net in Figure 4 (a).

The final lines requiring explanation are line 13 and lines 30 to 35. Line 13 introduces a new tube N_1. This tube is always empty ($N_1 =$) when the repeat-until loop enters a new iteration. Between line 30 to line 33, it depends on the values for $b1$ to $b4$, which strands (s_1, s_2, s_3, or s_4) are added to tube N_1. It is important to understand that at line 35, the Petri net went through one complete state transition (e.g., from Figure 4 (b) to Figure 4 (c)). Inside the repeat-until loop, the current "state" is always represented by the content of tube N_0, and the so-called "next-state" by that of tube N_1 in line 34. Line 13 empties tube N_1 in order to prepare it for the new next state. Line 36 decides whether the loop enters a new iteration. This depends on the new value for n, which was incremented in line 35.

The paper now goes through a couple of iterations to demonstrate the algorithm in more detail. A decision table (Table 1) keeps track of the boolean variables (which represent the behavior of the Petri net in terms of active places and content of tube N_0).

Column "Start" in Table 1 captures line one and two of the algorithm and is equivalent to the unmarked Petri net in Figure 4 (a). Column "Marking" represents line three to line ten in the algorithm and is equivalent to the marked Petri net in Figure 4 (b). Note two things here, first that strands are added to tube N_0 (line five and six), and second that this is reflected by corresponding settings by the boolean variables (line seven and eight). Note also that Table 1 indicates changes in boolean variable values (as the Petri net moves from one state to the next) by underlining these values. For instance, from the "Start" state to the "Marking" state in Table 1 the values for $b1$ and $b3$ change and therefore are underlined. Anyhow, at line ten the settings are $b1 = true$, $b2 = false$, $b3 = true$, and $b4 = false$.

			Iteration, repeat loop					
	Start	Marking	1	2	3	4	5	6
$b1$, *detect*$(N_0(s_1))$	0	$\underline{1}$	$\underline{0}$	0	$\underline{1}$	$\underline{0}$	0	$\underline{1}$
$b2$, *detect*$(N_0(s_2))$	0	0	$\underline{1}$	$\underline{0}$	0	$\underline{1}$	0	0
$b3$, *detect*$(N_0(s_3))$	0	$\underline{1}$	1	$\underline{0}$	$\underline{1}$	1	0	$\underline{1}$
$b4$, *detect*$(N_0(s_4))$	0	0	0	$\underline{1}$	$\underline{0}$	0	$\underline{1}$	$\underline{0}$
State similar to Figure 4	(a)	(b)	(c)	(d)	(b)	(c)	(d)	(b)

Table 1: Decision table, illustrating the behavior of the Petri net in Figure 4.

Next, variable n is set to nil in line 11. According to the values for $b1$ to $b4$, the repeat loop enters the first if-then statement (line 14) only. Consequently, when the index n is incremented in line 35 then $b1 = false$ (line 15) and $b2 = true$ (line 16), whereas $b3$ and $b4$ remain unaltered ($b3 = true$ from line eight, and $b4 = false$ from line two). So, the settings after the first iteration are $b1 = false$, $b2 = true$, $b3 = true$, and $b4 = false$. This state is equivalent to Figure 4 (c).

In the second iteration these settings activate the second if-then statement only ($b2 = true$, and $b3 = true$). Consequently, $b2 = b3 = false$ (line 20), $b4 = true$ (line 21), and variable $b1$ remains unaltered (*false*). Now, the settings are $b1 = false$, $b2 = false$, $b3 = false$, and $b4 = true$. This state is equivalent to Figure 4 (d).

In iteration three, these settings activate the third if-then statement (line 24) only ($b4 = true$). Therefore, $b4 = false$ (line 25), $b1 = true$ (line 26), $b3 = true$ (line 27), and $b2$ remains unaltered $false$. So, the settings are $b1 = true$, $b2 = false$, $b3 = true$, and $b4 = false$. This state is equivalent to Figure 4 (b) again, and the Petri net process illustrated in Figure 4 starts again. Table 1 captures a few more iterations. Playing these iterations through demonstrates that the algorithm indeed models the behavior of the Petri net in Figure 4, however, this also indicates that the paper achieved its main goal.

4 Discussion

The previous sections successfully led to our goal, the modeling of Petri nets based on DNA computing principles. It is necessary, however, mentioning that the algorithm presented here is an ad hoc solution to the problem. It is possible, for instance, to use only one tube N_0, and to write the algorithm without any of the four boolean variables by replacing them with corresponding biochemical operations. For example, imagine the state in Figure 4 (c) equivalent to strand s_2 and strand s_3 in tube N_0. Now, imagine a progression to Figure 4 (d) where strands s_2 and s_3 need to be separated from tube N_0 and strand s_4 added to the same tube. This could be achieved by the following biochemical operations, *separate*$(-N_0, s_2)$, *separate*$(-N_0, s_3)$, and *add*(s_4, N_0). We found, however, statements such as $b2 = false$, $b3 = false$, etc. much simpler to handle and follow, helping a reader to better understand the logic of the algorithm, and also providing an easier mapping between the logic of the algorithm and the behavior of the Petri net in Figure 4.

Another possible modification relates to the modeling of active places. Currently, an active place is modeled by a single DNA strand, and the firing conditions for a transition are determined by checking for the activity of its input places (i.e., corresponding DNA strands). Line 19 in the algorithm, for example, checks for the two individual stands s_2 and s_3 in tube N_0. Another way would be to model a transition by connecting all its active places into a single strand. The Watson-Crick complement can be used for connecting two stands in a pre-defined fashion. One possibility would be to connect s_2 and s_3 via the strand $(e_{2\to3})$, where $(e_{2\to3})$ is the Watson-Crick complement of the second half of s_2 concatenated with the Watson-Crick complement of the first half of s_3. If the complements for s_2 and s_3 are s'_2 and s'_3, respectively, then we may have the example illustrated in Figure 6 (a), and in case strands s_2, s_3, and $e_{2\to3}$ where mixed together in a tube (e.g., N_0) then the double strands illustrated in Figure 6 (b) might be generated by bonding.

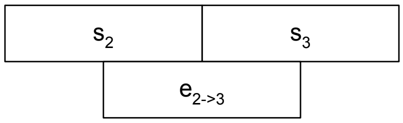
It is now possible to write an algorithm including some of the biochemical procedures mentioned in Section 3.1, as well as others, to check for the existence of strand s_2s_3 in tube N_0. If the strand exists then the code executed after may be similar to that following line 19 in Figure 5. The paper does not go into further details here about biochemical procedures or the algorithm reflecting these procedures, but the reader is directed to Adleman's (1) work, which entails details that are very similar to the facts just mentioned.

In terms of other issues, there is also the fact that the presented work deals with a single example only. Although the example may not really bring to the fore the great advantage DNA computing provides, namely parallelism, it is not difficult to envisage two or more Petri nets running in parallel and interacting amongst each other (e.g., interactions may be messages exchanged in the form of tokens). This should not devalue the paper, because the major contribution of in this work is the synergy of two fields—Petri nets and DNA computing. A final though considers the purely theoretical treatment of the subject. Such a treatment should not suggest any ignorance of the many challenges DNA computing still poses for engineers working in a broad variety of disciplines involved with DNA computing.

(a)

s_2 = TATCGGATCGGTATATCCGA
s_3 = GCTATTCGAGCTTAAAGCTA
s_2s_3 = TATCGGATCGGTATATCCGAGCTATTCGAGCTTAAAGCTA
s'_2 = CATATAGGCT
s'_3 = CGATAAGCTC
$e_{2\to3} = s'_2s'_3$ = CATATAGGCTCGATAAGCTC

(b)



TATCGGATCGGTATATCCGAGCTATTCGAGCTTAAAGCTA
CATATAGGCTCGATAAGCTC

Figure 6: Alternative modeling of transitions and places.

5 Summary

The paper suggests Petri nets as a novel DNA computing application area. The paper demonstrated the feasibility of the approach in theory. The paper indicates that real-life applications of the presented work may be problematic to achieve because of various engineering challenges in the field of DNA computing. This does not mean, however, that the approach could not be verified in vitro in a DNA computing project.

References

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

[2] Chang, W. and Guo, M. (2003). Solving the set-cover problem and the problem of exact cover by 3-sets in the Adleman-Lipton's model. *BioSystems*, 72(3):263–275. Elsevier Science.

[3] Chang, W. and Guo, M. (2004). Molecular solutions for the subset-sum problem on DNA-based supercomputing. *BioSystems*, 73(2):117–130. Elsevier Science.

[4] Heiner, M., K. I. and Willa, J. (2004). Model validation of biological pathways using Petri nets–demonstrated for apoptosis. *Biosystems*, 75:15–28. Computational Systems Biology.

[5] Kounev, S. and Buchmann, A. (2006). SimQPN–a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4–5):364–394.

[6] Lipton, J. (1995). DNA solution to hard computational problems. *Science*, 268:542–545.

[7] Paun, G., R. G. and Salomaa, A. (1998). *DNA Computing–New Computing Paradigms*. Springer-Verlag, N.Y.

[8] Peterson, J. (1981). *Petri Net Theory And The Modelling Of Systems*. Prentice Hall, Inc., Englewood Cliffs, N.J.

[9] Petri, C. (1961). *Kommunikation mit Automaten*. PhD thesis, University Bonn, Germany. PhD thesis.

[10] Petri, C. (1982). State-transition structures in physics and in computation. *International Journal of Theoretical Physics*, 21(12):979–992.

[11] Reza, H. (2006). A methodology for architectural design of concurrent and distributed software systems. *The Journal of Supercomputing*, 37(3):227–248.

[12] Schuster, A. (2003). DNA algorithms for rough set analysis. In Liu, J., Cheung, Y.M, and Yin, H., editors, *Intelligent Data Engineering and Automated Learning*, volume 2690 of *Lecture Notes in Computer Science*, pages 498–513. Springer-Verlag, Berlin.

[13] Schuster, A. (2005). DNA databases. *BioSystems*, 81(3):234–246. Elsevier Science.

[14] Soreni, M., Yogev, S., Kossoy, E., Shoham, Y., and Keinan, E. (2005). Parallel biomolecular computation on surfaces with advanced finite automata. *J. Am. Chem. Soc. (Article)*, 127(11):3935–3943.

[15] Watson, J. and Crick, F. (1953). Molecular structure of nucleic acids. *Nature*, 171:734–737.

[16] Zhu, P. and Schnieder, E. (2000). Holistic modeling of complex systems with Petri nets. In *Proceedings IEEE International Conference on Systems, Man, and Cybernetics (SMC'2000), 8-11 October 2000, Nashville, TN*, volume 4, pages 3075–3080.