Computational Biology: A Programming Perspective

Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen,
and Søren Bjerregaard Vrist

Department of Computer Science (DIKU), University of Copenhagen, Denmark
{hartmann,neil,simonsen,seet}@diku.dk
DIKU URL: http://www.diku.dk

Abstract. Computation via biological devices has been the subject of close scrutiny since von Neumann's early work some 60 years ago. In spite of the many relevant works in this field, the notion of *programming* biological devices seems to be, at best, ill-defined. While many devices are claimed or proved to be computationally universal in some sense, the full step to a bona fide programming language is rarely taken, and one question is noticeable by its absence: If the device is universal, *where are the programs?*

We begin with an extensive review of the literature on programming-related biocomputing; and briefly identify some strengths and shortcomings from a programming perspective. To show concretely what one could see as programming in biocomputing, we outline (from recent work) a computation model and a small programming language that are biologically more plausible than existing silicon-inspired models. Whether or not the model is biologically plausible in an absolute sense, we believe it sets a standard for a biological device that can be both universal *and programmable*.

1 Context

The terms "biocomputing" and "systems biology", taken in their broadest senses, span many fields and areas of research: biology, chemistry, physics, mathematics, electrical engineering and computer science among others. Two major subareas:

- Computer modeling of biological and biomolecular processes
- Biological hardware for computation

The terms "biomolecular computation" and "biomolecular computational model" occur with both meanings in the literature (see Section 4), sometimes referring to the one and sometimes the other subarea. This paper emphasises the second. More precisely: we take a *synthetic* viewpoint, concerned with building things as in the engineering and computer sciences. This is in contrast to the inevitable and ubiquitous *analytic* viewpoint common to the natural sciences, concerned with finding out how naturally evolved things work.

G. Agha, O. Danvy, and J. Meseguer (Eds.): Talcott Festschrift, LNCS 7000, pp. 403–433, 2011.

We ask: What can be *done or built or constructed*; and not: how does nature work? (Caveat: just as in engineering, one will need to understand nature's cause-and-effect sufficiently well to be able to *modulate* it, i.e., to use nature to effectively serve our purposes.)

1.1 A Theme: Biological Problem-Solving

Our main aim is thus to use the biological world as a computational medium. From a programming perspective, the main goal is problem-solving by writing programs. This section thus refers to solving problems *by biology*, not solving problems *about biology* or *in biology*.

Given: a computational problem. **To find:** a biological solution. Further, problem-solving by a program means finding *a general solution* (and not just one run of one algorithm on one input data instance).

A top-down approach is to devise a model of computation that

- satisfies requirements for computer science/engineering
- that could conceivably be realised in a biological medium
- in which programs are clearly visible, and programming can be done
- is a framework in which it is possible, at least in principle, to say when a problem has been solved

A bottom-up approach is to develop *biological toolkits and environments* in which such engineering can be done. Examples include: synthetic biology; DNA computing; membrane computing; and other computation models, e.g., peptides, whole cell computing, bio quantum computing and many others seen in the literature. Some of these toolkits and environments are already well-started and others just beginning.

A reservation: our impression is that many exciting papers have been written on computational biology, but there seem to be relatively few and small actual realisations with concrete, reproducible, automatable results. There are many possible reasons for this: intractability of physical biological media, difficulty of carrying out experiments, difficulty of measuring success, liberal funding, journalistic appeal, etc.

1.2 Anticipated Difficulties in Practical Computational Biology

Our main goal is to find a way to incorporate programming concepts into a biological context. In this paper we will sidestep some anticipated difficulties including

Instrumentation: In a laboratory setting (with test tubes, Petri dishes, chemical solutions, ...) one would need to

1. put an experimental set-up into a known initial state
2. put input values into it
3. recognise when a computation has finished (or run "long enough")
4. read output values out from it

Although these challenging problems are important, our approach is synthetic and top-down. Future work would be to integrate our approach with a bottom-up approach, and to realise the ideas in a biochemical laboratory.

Stochastic factors: Another dimension of difficulties that we will abstract away from is how to deal with the nondeterminism that seems built-in to nature, for example Brownian motion, noise (in an engineering sense), quantum state transitions, and many others.

Researchers have worked to alleviate these real problems, studying error-correcting processes from a theoretical viewpoint, and in nature as well (e.g., DNA repair).

1.3 About the Scientific Method and Automatability

From the analytic viewpoint: natural science is strongly based on the principle of *reproducibility*: if someone reports something in a scientific report, it should be possible for the readers (if sufficiently determined) to reproduce the results by running the same experiments themselves. This approach has had many successes, and some highly visible failures (e.g., cold fusion).

From the synthetic viewpoint: reproducibility is *relatively easy to achieve* in computer science. Computers have been carefully designed and built to be deterministic and predictable (else there is a bug in the computer's construction), so reproducing a result can consist of just running the program again.

Reproducibility is *much harder* in biological frameworks. There is typically no program to run, but rather long and complex experimental processes involving significant hand work, and human participation and/or evaluation. Further, randomness is real: what works in one experimental setting may not work in another, for reasons mentioned above. Much research has been aimed at just this problem: ensuring predictable behavior; and many reported experimental results seem to suffer from it.

A measure of progress in both analytic and synthetic work is *automatability*. A well-known real-world example: the rapid advances made in DNA analysis, e.g., for identification of people in legal or criminal settings. In computer science and engineering, the goal is a framework to make experimentation automatable so an experiment can be carried out a hundred or a thousand times. Further, there is an a priori desired known relation between input and output of the experiment: a relation that can be reliably expected to hold in all conceivable instances of just this experiment (else the hypothesis/principle/program is insufficient, incomplete, or just plain wrong).

This approach stands in great contrast to the analytical framework. Phrased somewhat strongly: if the expected output is fully known in advance, one is working not on an experiment, but on development rather than research. (To be fair: the synthetic approach is aimed at development, not at discovering new scientific truths.)

2 Motivation

2.1 Biocomputation

An initial hope biocomputing was to overcome some of the limiting factors of conventional microchip-based computers, similar to the hopes for quantum computation: can we break through the barrier of polynomial running time [5,1,17]?

A dream: instead of a computer built from microchips and based on electrical signals, a biocomputer would run on "hardware" based on biological materials with some biochemical or biological equivalent of electrical signals. The hope is that a completely new paradigm might yield an advantage over conventional computing by putting radically different constraints on computation. Advantages may arise from the mere difference of scale of materials, and complexity of computation possible for basic blocks of a biomolecular computer.

Research in the 1990s focused on exploring such possibilities. Adleman made a groundbreaking proof of concept in 1994: a DNA based computer that solved an instance of a special case of the traveling salesman problem with 7 cities [1]. Several other possible advantages of a biocomputer were theorized—including biology-inspired properties such as autonomy, self-assembly, self-repair, high information density [57,54], and advantages due to inherent parallellism:

> In a world where parallel computing is in focus, molecular interactions like communication over noisy media and load balancing are inherent in the structure of molecules refined by evolution.
>
> Garzon and Deaton, [57]

As many different molecules can be synthesized at low cost and exist abundantly around us [57], natural resources are plentifully available for biocomputation. Another great potential lies in the fact that a biocomputer can in principle interact directly with other biological material, including the cells in the human body [11]. Numerous researchers [23,114,9,7] mention the possibility for direct interactions with live cells in one way or another.

Some Remarks by Benenson [11] First, on the issue of generality:

> Unlike a silicon computer that can be reprogrammed with a few hits on a keypad, a biomolecular computer is really a set of design tools which, when provided with a description of a computational task, generates a blueprint of a molecular network that can implement this task. One important challenge is to make sure that these tools are flexible enough to enable a sufficient variety of tasks.

Second, on universal biological machines:

> In theoretical computer science this problem was solved by the invention of universal models of computation, such as the Turing machine.

> Although it seems unlikely that similarly universal approaches could be realized with biomolecular building blocks anytime soon, biocomputer architectures could, and should, aspire to at least some degree of generality in the computational sense.

2.2 Biomolecular Algebras and Calculi

Formalisms have been devised to do computer modeling (from an analytical viewpoint) of observed biomolecular processes. Examples include the κ-calculus [39,36], BioAmbients [100], Biochemical Ground form [26,25], Strand Algebra [22], and Pathway Logic [123,31].

A natural next step is to think of these formalisms synthetically, e.g., to reduce the need for expensive laboratory experiments.

2.3 Programming

In research on biomolecular computation the words "programming" and "calculus"are often used; for example, [65,9,57,96] all mention programming as part of their title. *This is not the same as the "programming" familiar to computer scientists: writing a program in a programming language.* Instead, the papers cited use the word "programming" to describe the process of designing biological devices in a setup that will carry out *one* computation or simulation of biomolecular interactions, and not a generic program that can be run on many inputs.

Program Text 1.1. Program that appends two lists, in Standard ML syntax

```
fun append (a::as) bs  = a:: append as bs
  | append [] bs = bs
```

For example, take a simple "append" program as in Program text 1.1. Any computer scientist would recognize this as a program. Compare this with, for example, the κ-calculus from [39,36]:

> κ is a formalism for modeling molecular biology where molecules are terms with internal state and with sites, bonds are represented by names that label sites, and reactions are represented by rewriting rules. For example, $EGFR[tk^0](1^z)$ represents a molecule of species $EGFR$ that is not phosphorylated - the internal state tk is 0 - and that is bond to another molecule - its site 1 is labeled with a name z.

and a κ rule as seen in Figure 1. In our opinion, the connection to conventional programming, in the sense of solving a problem or writing an algorithm, is not obvious.

$$EGF(1^x + 2^y), EGF(1^x + 2^z), EGFR(1^y), EGFR[tk^0](1^z)$$
$$\rightarrowtail EGF(1^x + 2^y), EGF(1^x + 2^z), EGFR(1^y), EGFR[tk^1](1^z)$$

Fig. 1. κ rule defining the first step of *Receptor Tyrosine Kinase* from [44]

Figure 2 shows another example, used by Yin et al in [133] to illustrate "Pathway programming". Here, the "programming" is in the process of designing the DNA to react and function in a specific way, and is *not* analogous to Program text 1.1.

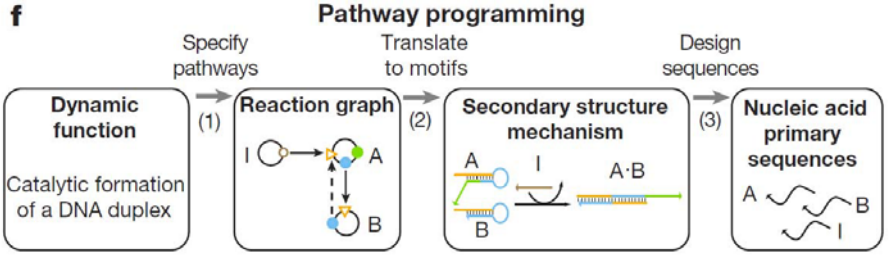


Fig. 2. From [133]. Images showing steps to construct a "molecular executable".

Figure 3 illustrates the molecular implementation of a propositional logic query as presented by Ran, Kaplan and Shapiro in [98]. Both the "program" and the input are represented as specific molecular "objects":

> A compiler translates facts, rules and queries into their molecular representations and subsequently operates a robotic system that assembles the logical deductions and delivers the result.(Ran, Kaplan and Shapiro [98])

This version of biomolecular programming is closer to the idea of programming as in Program text 1.1, but it still conflicts with a key concept of conventional programming, e.g., as we say later:

> A program is software, not hardware. Thus a program should itself be a concrete data object that can be replaced to specify different actions.

The central question: can program execution take place in a biological context? Evidence for "yes" includes many analogies between biological processes and the world of programs: *program-like behavior*, e.g., genes that direct protein fabrication; "switching on" and "switching off"; processes; and reproduction.

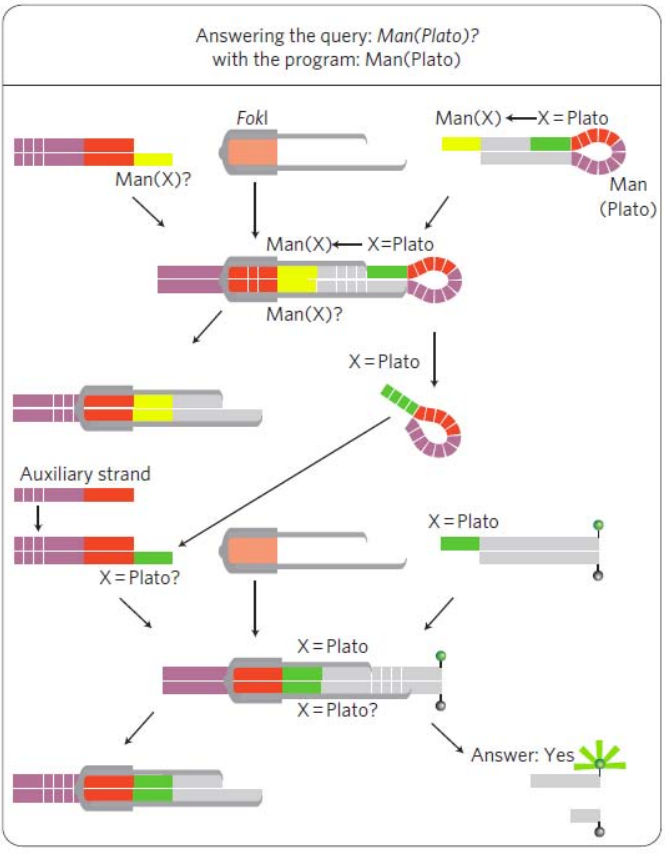


Fig. 3. Ran, Kaplan and Shapiro [98]: the molecular representation of a simple logic query

3 Biochemical Universality and Programming

We begin in Section 4, by evaluating some established results on biomolecular computational completeness from a programming perspective; and then constructively provide an alternative solution in Section 5. The new model seems biologically plausible, and usable for solving a variety of problems of computational as well as biological interest.

3.1 Baseline: Program Execution

What do we mean by a program (roughly)? An answer: a *set of instructions* that specify *a series (or set) of actions on data*. Actions are carried out when the

instructions are executed (activated,...) Further, a program is software, not hardware. Thus a program should itself be a *concrete data object that can be replaced* to specify different actions.

Program execution: we write $[\![\texttt{program}]\!]$ to denote the meaning or net effect of running `program`. A program meaning is often a function from data input values to output values. The `program` is *activated* (run, executed) by applying the semantic function $[\![\_]\!]$. The *task of programming* is, given a desired semantic meaning, to find a program that computes it. Some mechanism is needed to execute `program`, i.e., to compute $[\![\texttt{program}]\!]$. This can be done either by hardware or by software.

3.2 Turing Completeness of Computational Models

Turing completeness of a computation framework is typically shown by *reduction* from another problem already known to be Turing complete. Notation: let L and M denote languages (biological, programming, whatever), and let $[\![p]\!]^L$ denote the result of executing L-program p, for example an input-output function computed by p. Say that language M is at least as powerful as L if

$$\forall p \in L-\text{programs } \exists q \in M-\text{programs } (\ [\![p]\!]^L = [\![q]\!]^M\ )$$

A popular choice (for proving universality) is to let L be some very small Turing complete language, for instance Minsky register machines or two-counter machines (2CM). The next step is to let M be a biomolecular system of the sort being studied. The technical trick is to argue that, given any L-instance of (say) a 2CM program, it is possible to construct a biomolecular M-system that faithfully simulates the given 2CM. This discussion brings up a central issue:

Simulation as Opposed to Interpretation. Arguments to show Turing completeness are (as just described) usually by *simulation*: For each problem instance (say a 2CM program) one constructs (using any available method) a biomolecular system that solve the problem. However in many papers the construction of the simulator is done by hand by the author, and each problem instance require a new hand coded simulator. In effect the existential quantifier in $\forall p \exists q([\![p]\!]^L = [\![q]\!]^M)$ is computed by hand. This phenomenon is clearly visible in papers on cellular computation models: completeness is shown by simulation rather than by interpretation.

In contrast, Turing's original "Universal machine" realises p's computation by means of *interpretation*: a stronger form of imitation, in which the existential quantifier is realised by machine. Turing's "Universal machine" is capable of executing an arbitrary Turing machine program, once that program has been written down on the universal machine's tape in the correct format, and its input data has been provided. Our research follows the same line, applied in a biological context: we show that simulation can be done by general interpretation, rather than by one-problem-at-a-time constructions as mentioned by Benenson in [11], quoted in Section 2.1 of this paper.

3.3 Programs in a Biochemical World

Our goal is to express programs in a biochemical world. Programming assumptions based on silicon hardware must be radically re-examined to fit into a biochemical framework. We briefly summarize some qualitative differences.

- **There can be no pointers to data,** i.e., no addresses, links, or unlimited list pointers. In order to be acted upon, a data value must be *physically adjacent* to some form of actuator. A biochemical form of adjacency: a chemical bond between program and data.
- **There can be no action at a distance**: all effects must be achieved via chains of local interactions. A biological analog: *signaling.*
- **There can be no nonlocal control transfer**, e.g., no analog to GOTOs or remote function/procedure calls. However control loops can be accepted, provided the "repeat point" is (physically) near the loop end. A biological analog: *a bond* between different parts of the same program.
- On the other hand there exist available **biochemical resources** to tap, i.e., free energy so actions can be carried out, e.g., to construct local data, to change the program control point, or to add local bonds into an existing data structure. Biological analogs: Brownian movement, ATP, oxygen.

The first three points above are very different from basic architecture currently used in a silicon based computers: the addressing unit and the address decoder are built around the ideas of unbounded pointers to both data and program: "random access memory" and control transfers.

4 History and Literature Review

4.1 History

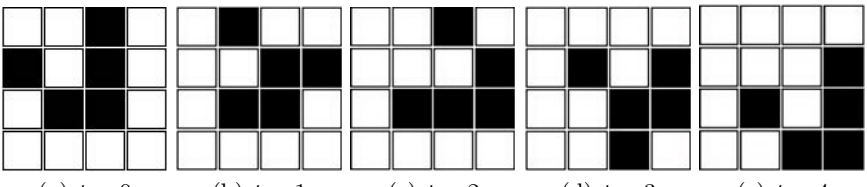
Our brief survey of historical developments is not in any way exhaustive, but should "scratch-the-surface" enough to give a feeling for the different aspects of the field. Some good overviews for further reading include Benenson [10], and Kari and Rozenberg [76,78].

Background. Nature-inspired models of computation can demonstrate both how nature inspires computation models, and how complexity can emerge from seemingly simple rules in nature. Examples include cellular automata, neural computation, evolutionary programming, swarm intelligence, artificial life, membrane computing, and amorphous computing [78].

Cellular Automata
These were considered in the early 1950s as a possible idealized model of biology [131, page 48]. The work of Von Neumann and Burks on a "universal constructor" [93] was inspired by self-reproduction both for biology and computers.

Roughly, a cellular automaton is a grid of cells, typically 2-dimensional, in which each cell is in one of a fixed finite set of states. First, the initial state ($t = 0$) is set up by choosing an initial assignment of states to cells. The next state of any cell, at time $t+1$, is determined by applying a predetermined transition function to current state of the given cell and its immediate neighbors. For example "Any white cell with exactly three black-neighbors becomes a black cell."[1].

The perhaps most famous cellular automaton is *Game Of Life*, the two-state two-dimensional cellular automaton by John Conway [55] which has exactly four rules. Figure 4 illustrates[2] the initial generation and four following generations of a *Game Of Life* cellular automaton. In [15] it is shown that universal computation is possible with the *Game Of Life*, and an actual implementation was demonstrated in [28]. Concretely, this means that for any Turing machine, an initial state exists, from which the *Game Of Life* will faithfully simulate the Turing machine.



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$ (e) $t = 4$

Fig. 4. Five steps of a *Game Of Life* life form called "glider" [15]. Notice that the last image contains the same shape as the first image. This means that the glider will continue to fly until it hits something.

Cellular automata are homogeneous, parallel, and all interactions are local. Further, it is possible to obtain natural physical properties such as reversibility and conservation laws by choosing local update rules properly [76]. This connection would be especially interesting if it is possible to let nature simulate cellular automata, and thereby prove that nature can obtain the Turing completeness properties as demonstrated for cellular automata [76]:

> ...then perhaps we eventually succeed to harness physical reactions of microscopic scale to execute massively parallel computations by running a computationally universal CA. ... While such truly programmable matter may be decades away, its potential is great

Wang Tiles

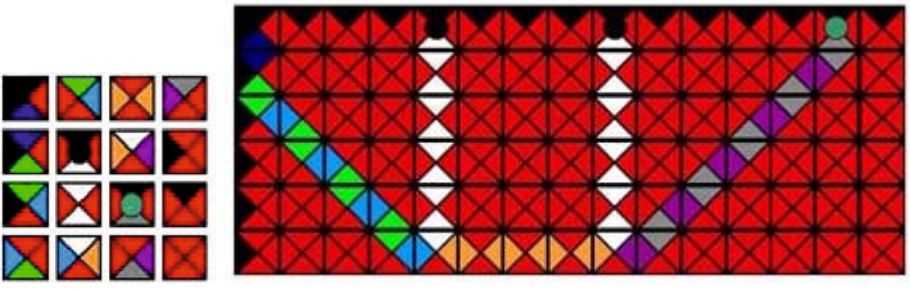
These were introduced by Hao Wang in 1961 [125]. A tiling system is a finite set of tile types. The tiles are equal-sized squares, with a color on each edge. A tiling is a two-dimensional arrangement (without rotation or reflection of the tiles) of

[1] Rule from *Game Of Life*.

[2] Images generated via `http://www.onderstekop.nl/GoF.php` - 2010-02-07

tiles, such that the touching edges of adjacent tiles must have the same color. Figure 5(a) shows a set of Wang tiles and 5(b) shows a tiling. It is undecidable, given a finite set of Wang tiles, whether they can tile the plane [14]: any Turing machine can be emulated by Wang tiles in such a way that if the tiles tile the plane, the Turing Machine never terminates [103].

Figure 5(b) shows[3] an encoding of Wang tiles that represents the calculation of $5 + 9$ with the result of 14.



(a) Wang tiles used

(b) The tiles types assembled into a pattern representing $5+9 = 14$. Notice the three special tiles with the circle in the middle in the top row. The two black ones represent the input numbers, and the green one represents the answer. For this set of tiles, addition with input and output is the only possible way to assemble them.

Fig. 5. Based on a diagram from "Tilings and Patterns" by Grunbaum and Shephard

Biocomputers. The idea of a biocomputer was first suggested by Feynman in 1959, and theoretically discussed through the eighties and beginning of the 1990s [54], for example in the visionary paper by Michael Conrad, "On design principles for a molecular computer" [33]. It was argued that the natural concurrency present in biocomputing could be used to solve difficult combinatorial problems, e.g., NP-complete problems.

In a landmark 1994 paper, Adleman demonstrated how a biological system can use reactions to solve a Directed Hamiltonian Path problem with DNA hybridization [1].

The advantage of such a "biocomputer" is potentially massive parallelism: the theoretical ability of DNA to contain data is 10^{21} bits per gram, and the energy requirement is low: $2 * 10^{19}$ operations are theoretically possible per joule [54]. Lipton found theoretical methods for solving NP-complete problems in general using DNA computers [82], and several enhancements were provided, e.g. Adleman et al discovered the solution to a 20-variable 3-SAT problem [18], and Winfree et al devised ways to use single strand DNA as memory [107,108].

Based on this research, several ways were devised to compute other search problems such as linear optimization problems [126], and Boolean search type satisfiability such as 3-coloring [27, sec 2.2]. The parallels between DNA-processing

[3] Images in Figure 5 from `http://seemanlab4.chem.nyu.edu/XOR.html`

reactions and "forbidding/enforcing systems" further led to the usage of DNA-processing to solve these kinds of problems as well [3]. DNA-based arithmetic was developed in [62,27]. Further developments include the breaking of DES encryption with DNA [17], DNA Computer based on Biochips [137], playing TicTacToe against DNA [90], DNA encoded with finite automata for medical purposes [9,7], and molecular implementation of simple logic programs [98].

However, while the generation of all paths in Adleman's original experiment took less than a second, it took weeks of manual lab work to extract the potential solutions from the DNA cocktail. The method is further hampered by the exponential growth of the size of the solutions: The weight of the DNA needed to represent the state space explored to find the solution of a 200 city Hamiltonian Path Problem would exceed the weight of the Earth itself [95].

Why DNA? Though the term "biocomputer" does not specify the specific computation medium, by far the most common medium is Nucleic Acids: the NA in DNA and RNA. This is possibly due to the fact that nucleic acids have a predictable "base pairing property" which makes them a powerful tool for biomolecular engineering [136]. In practice it seems that DNA and RNA behave "more deterministically" than other biological media (perhaps for evolutionary reasons).

Thus in the literature the terms "biocomputation" and "biocomputers" are often used interchangeably with "DNA Computer." (Exceptions: [11] uses RNA; and there are articles on "peptide computing", "whole cell computing" and "Quantum Molecular Computing": [71], [118], [34].) Following the general trend, in this section we will assume unless stated otherwise that a biocomputer is based on DNA.

4.2 Computational Universality of DNA Computers

Universality and self-reproduction. Several articles argue "universality" by showing (in several ways) that Turing machines can be simulated. Oddly, articles we have seen do not mention self-reproduction, even though DNA is the key to biological self-reproduction. Further, self-reproduction was a major goal of von Neumann and Burks [93]. (That pathbreaking work did indeed show how to simulate a universal Turing machine; but this was only incidental, as a means to establish that cellular automata did not "reproduce" in some trivial way, e.g., as in the growth of crystals.)

Several different approaches have been taken to represent Turing machine operations by DNA operations or collection of operations (e.g., hybridization, annealing, ligation or polymerization). A partial list follows. The different articles use different terminology, notations and degrees of formalism, so we choose to quote their claims directly, rather than paraphrase and interpret their results:

- [12,13]: Discuss the theoretical possibility of RNA/DNA: "The enzyme may thus be compared to a simple tape-copying Turing machine that manufactures" its output tape rather than merely writing on it.
- [119] : Shows how to create "a non-deterministic Turing machine" (one *specific* Turing Machine at that)

- [5] Discusses a design method to simulate a Turing Machine via a technique for "text-insertion" that "provides a basis for implementing general Turing Machines"
- [104] : " [we] propose an encoding for the transition table of a Turing machine in DNA oligonucleotides and a corresponding series of restrictions and ligations of those oligonucleotides that, when performed on circular DNA encoding of an instantaneous description of a Turing machine, simulate the operation of the Turing machine encoded in those oligonucleotides"
- [135] Proposes a new "splicing model" and "show that they have the same computational power as a Turing machine"
- [114]: A physical model (See figure 6) showing what a mechanical Turing machine on a biomolecular level could look like.
- [79,80] This reports that DNA mutagenesis "is theoretically universal by showing how Minsky's 4-symbol 7-state Universal Turing Machine can be implemented".
- [134] designs a complete "autonomous device capable of universal computation and universal translational motion"

Self assembly. Self assembly is a process whereby DNA can arrange itself in a shape without outside involvement. An example is the formation of double helix

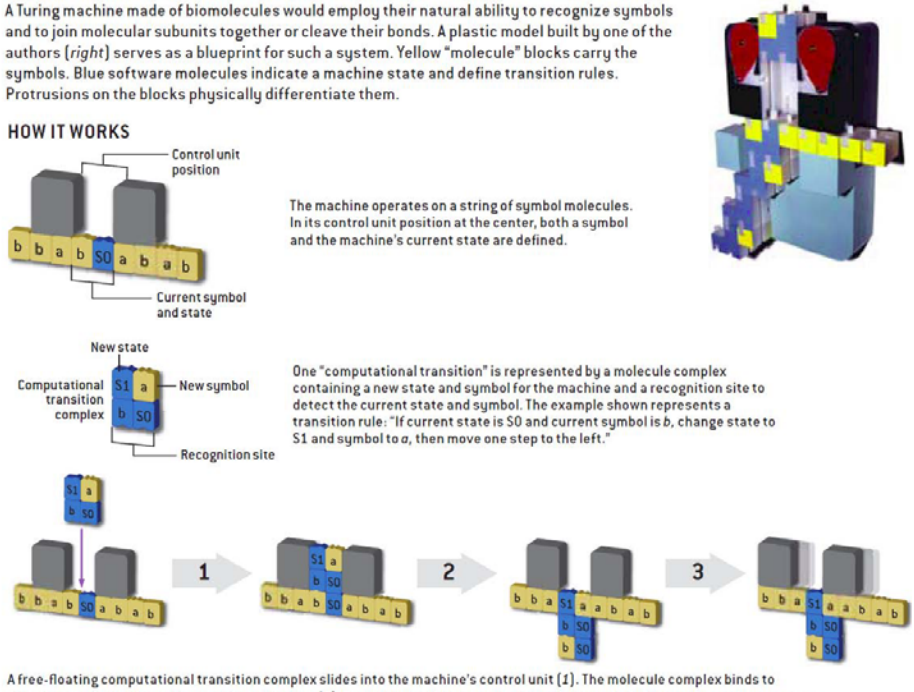


Fig. 6. From Shapiro and Benenson [113]

DNA from individual strands.It has been shown that DNA can do self assembly in a "programmable" way . Here "programmable" means that the final outcome of the self assembly process can be guided into an arbitrary two-dimensional shape, decided before the self assembly process begins.

The mechanism uses a concept of "sticky ends" of DNA structures [128,127]. An example is the famous nano scale "smiley" of [106].

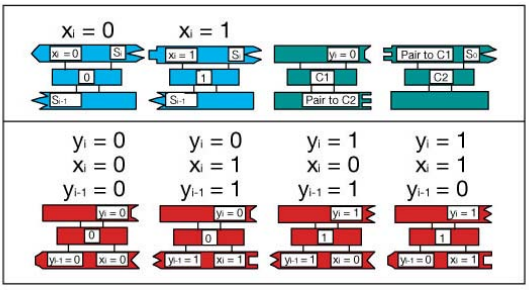
Based on the Wang tiling properties of DNA, [129] shows a way to simulate one-dimensional cellular automata, and [85] shows a way - via Wang tilings - to do a logical cumulative XOR. Figure 7 illustrates the principle of DNA tiling to realise a cumulative XOR gate. Figure 7(a) shows the building blocks used, and figure 7(b) shows how these can be tiled together to get the result of $Y = 1011$ when $X = 1110$.

Winfree and coworkers used and extended the similarity between Wang tiles and DNA with sticky ends. Using known abilities of Wang tiles, they did a proof for universal computation abilities within self assembly of DNA [129]. Even though the theoretical computational power is unlimited, the practical usage of these powers is hampered by the difficulty of controlling the geometry and the specificity of binding interactions [105].

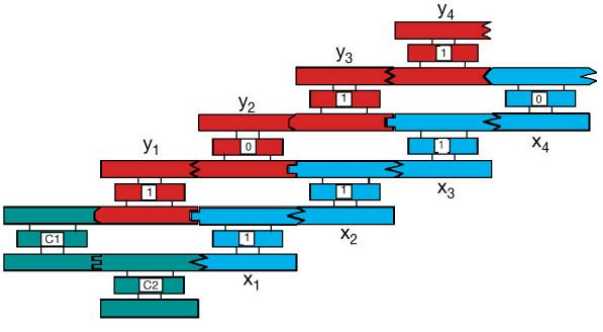
4.3 On Error Correction in Biological Contexts

In Section 1.2 we mentioned that "work has been done" to alleviate the stochastic factors when dealing with biomolecular computers. This section contains some references and quotes from the published research on the subject.

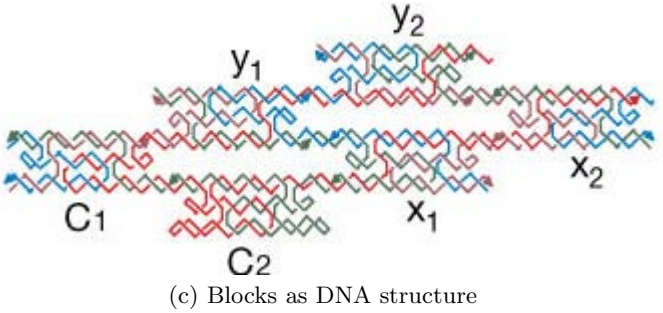
- Early work by Bennett [13,12] considered theoretical bounds in DNA computation: "The minimum error rate (equal to the product of the error rates of the writing and proofreading steps) is obtained when both reactions are driven strongly forward, as they are under physiological conditions."
- Chiniforooshan et al [29]: "We achieve this by designing a single-input, single-output restoration gate"
- Roweis et al [107]: "we discuss several methods for achieving acceptable overall error rates for a computation using basic operations that are error prone."
- Shlyahovsky et al [117] : "Finally, although the gate configuration consists of a complex structure composed of nine components, we find that the precise steps to set up the system and its activation at the defined conditions allow the result to be reproduced within a 15% error."
- Winfree at al [120]: "The correctness of our systematic construction was predicated on several idealizations of DNA behavior, and it is worth considering the deviations that we would expect in practice"
- Rothemund et al [104] discuss in detail possible errors.
- Reif and LaBean [102]Future Work "such as error correction and self-repair at the molecular scale"
- Murata and Stojanovich [91] discuss "error supression and correction"



(a) Translation of the a DNA with sticky ends to tile-like building blocks. Notice the different shapes and the ends of the blocks only allowing specific pairings of blocks. X_i blocks encode the input number, where Y_i encodes the output number, all written in the middle.



(b) The dna-blocks assembled with X_i blocks put together to represent $X = 1110$. As the blocks can only be assembled in a specific way the result in is encoded in Y_i as $Y = 1011$



(c) Blocks as DNA structure

Fig. 7. Images from [85]

4.4 Process Calculus and Formal Methodology

Process calculi are concerned with providing formal specifications of concurrent processes.

An early process calculus was CCS as defined by [70], later followed by much research including the π-calculus [86,110].

Several calculi have been proposed for biomolecular modeling, both to describe natural biological systems, and to model how the entities in these system interact. Process calculi are the "other side of things" in relation to DNA computers and biocomputing: how do we provide an abstraction for biomolecular systems that allows for quick and effective sharing, comparison, and correction of scientific knowledge? Regev and Shapiro [99] published a landmark paper postulating the design of a "language for the cell" built on the insights of the articles [51] (where Fontana et al use the λ-calculus to describe natural systems) and [101] (where Shapiro et al use the π-calculus for biological modeling). Proposed calculi include[4]:

- π-calculus for biology [101] as just mentioned.
- BioAmbients - Extension of the above bio-π-calculus [100].
- Brane Calculi - based on membranes considered as active elements, with the whole computation happening on the membrane [20].
- CSS-R - Formal biology done in CCS-R [38].
- Bio-PEPA - Extension of the known process algebra PEPA [58], designed for biochemical networks [30].
- κ-calculus [39,36].
- Biochemical Ground Form [26,25].

The basic idea is to provide means, via formal calculus, to describe and design biomolecular setups with formal rules. In a larger setting this field is known as "Systems Biology." At its core lie the fields of:

- Mathematical modeling, for example in terms of differential equations. This is a classical way to model biology.
- Computational modeling where the concepts of algebras [22,101,30], abstract interpretation [36,21] and process calculi [63] are used.

Computational modeling of biology, or the more apt "executable cell biology" [49], is a contemporary approach that uses computational power of conventional computers to reason about biomolecular systems.

Computational Power Often it is the case that these calculi are proven Turing complete, for example:

- π-calculus: proven by Milner by emulation of the λ-calculus [87].
- CCS [121].
- Brane Calculi [19].

[4] Based on the survey by [63].

– κ-calculus [36].

As noted by Fontana et al. ([36] page 2), the fact that a biological representation language is Turing complete need not imply that biological reactions per se are Turing complete.

Applications to the building "biocomputers" or parts thereof is a popular recent research direction. For example, [22,96] use a theoretical algebra to as a tool for defining and designing biomolecular gates for specific setups. This provides a way to design a "biomolecular computer" from the basis of Boolean networks or Petri nets. From there, they compile into an intermediate "Strand Algebra" that can be translated further into a specific DNA Strand mixture. The authors claim that this mixture could potentially simply be mail-ordered from a bio company, at relative low cost.

That approach effectively combines the area of biocomputing with the techniques (and benefits) of biomolecular calculi. A computer scientist's way to look at these biomolecular calculi could be as "a domain specific language for modeling biomolecular "things," rather than a general purpose programming language like Python, C, or Java. The same connection can be seen in the title of [45] - A Domain-Specific Language for Programming in the Tile Assembly Model - for modeling "tiles" which can be used to direct DNA self-assembly into tile structure (similar to Winfree's approach [127]).

4.5 Recent Developments

In recent years, it has become evident that biocomputing is perhaps not the answer to obtain fast solutions to computationally intractable problems (Parker, Cardelli [95,24]). Nevertheless, other potentials of a biocomputer in some form still interesting, e.g., self-replication, self-correction, massive parallelism, minuscule size, filters, ready availability, interfaces with other biological elements like humans. Several techniques to exploit the computational power of biomolecular systems are being explored, including cross-over from nano-research for tiling self-assembly [132,127], and a wide variety of specific proof-of-concept setups performing some specific computations.

Some recent research focuses on the abilities to create "biomolecular gates" in some form or the other. For example

– Papers [112,120,97,111,29,136] propose designs for DNA Strand displacement based gates (sequence based, enzyme free). Paper [29], building on the others, provides a design with four "desirable properties: scalable, time-responsive, energy-efficient and digital." [29]
– Papers [117,48] manually design DNA gates for AND, OR, XOR by "DNA scaffolding", report on the construction of a DNA based "library" of "DNAzymes", and demonstrate how to create composite gates as well, partly based on the DNA Strand Displacement techniques as described above.

- Paper [60] proposes a way to create logic gates (enzymatic) that can be used repeatedly, as opposed to the use-once gates of DNA Strand Displacement as mentioned in the previous two items.
- Paper [11] reviews recent advances for using RNA as logic gates and the ability to do "molecular logic". It describes the usage of RNA for computation in yeast and mammalian cells and notes that future work needs to address the bioengineering foundations of building these systems.
- Paper [81] reports on a successful coupling of molecular based logic gates, with silicon based microchips used to digitally read results of biomolecular computations. They claim that "the developed systems are the first examples of enzyme-based biocomputing systems interfaced with ordinary silicon-based electronics."

With biological gates at hand it should be possible to develop "wetware hardware" for executing computations but "the state of the art in biomolecular circuits remains far behind its electronic equivalent"[29].

In [98] Ran, Kaplan and Shapiro take the design of logic gates a step further and use the gates for specific "systems". Paper [98] addresses the issue that in general, all encoding and decoding of input and output is done manually in the lab for each problem, by developing a "compiler" that translates first-order logic statements and queries into the specific DNA strands that encode the input and the gates needed for calculating the result. Figure 3 shows a translation of a logic query into DNA structures.

In another direction, Winfree et al in [84] show how to use DNA origami [106] to direct a "molecular spider" to autonomously carry out sequences like "start,"follow", "turn" and "stop". This could be imagined to be used as a way to implement "Turing-universal algorithmic behavior"[84].

4.6 Conclusion

Research in the area of biocomputing, biomolecular algebras/calculi has been massive. Although still very much work in progress, the possibility of biomolecular computers holds great promise. That being said, even though some concepts and words known from computer software engineering, and computer sciences, are also seen in biocomputer design and research, a gap remains to be filled. The ultimate goal for a programming language on a biomolecular platform would be the ability to abstract away unnecessary details about the biological "hardware" and to be able to focus on writing algorithms and solving problems. The programming niche of the field of biomolecular computer research is still, as indicated by this literature survey, very much relevant and mostly unexplored.

Our work is a step towards bridging the aforementioned gap.

5 The Blob Model of Programmed Universal Computation

We take a very simplified view of a (macro-)molecule and its interactions, with abstraction level similar to the Kappa model [40,26,44]. To avoid misleading de-

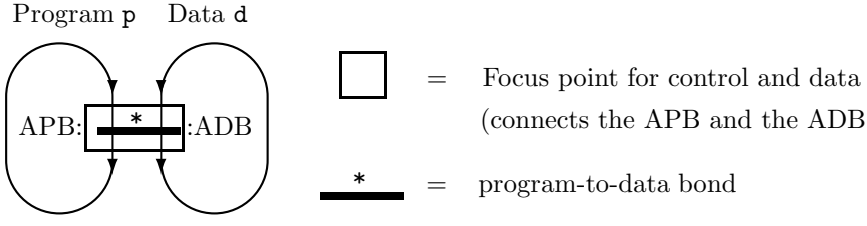
tail questions about real molecules we use the generic term "blob" for an abstract molecule. A collection of blobs in the biological "soup" may be interconnected by two-way *bonds* linking the individual blobs' *bond sites*.

A *program* p is (by definition) a connected assembly of blobs. A data value d is (also) by definition a connected assembly of blobs. At any moment during execution, i.e., during computation of $[\![p]\!](d)$ we have:

- *One* blob in p is active, known as the *active program blob* or APB.
- *One* blob in d is active, known as the *active data blob* or ADB.
- A bond `*`, between the APB and the ADB, is linked at a specially designate bond site, bond site 0, of each.

The main idea is to keep both program control point and the current data inspection site always close to a *focus point* where all actions occur. This can be done by continually shifting the program or the data, to keep the *active program blob* (APB) and *active data blob* (ADB) always in reach of the focus. The picture illustrates this idea for direct program execution.

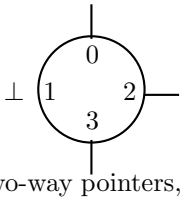
Running program `p`, i.e., computing $[\![$`p`$]\!]$(`d`)



The data view of blobs: A blob in our model have *four bond sites*, identified by numbers $0, 1, 2, 3$. At any instant during execution, each can hold a bond – that is, a link to a (different) blob; or a bond can hold $\perp$, indicating unbound.

In addition each blob has 8 *cargo bits* of local storage containing Boolean values, and also identified by numerical positions: $0, 1, 2, \ldots, 7$. (A biological analog to bits 1 or 0 is "phosphorylated" or "unphosphorylated".)

A blob with 3 bond sites bound and one unbound:



Since bonds are in essence two-way pointers, they have a "fan-in" restriction: a given bond site can contain at most one bond (if not $\perp$).

The program view of blobs: Blob programs are sequential. There is no structural distinction between blobs used as data and blobs used as program. A single, fixed set of instructions is available for moving and rearranging the cursors, and for testing or setting a cargo bit at the data cursor. Novelties from a computer science viewpoint: there are no explicit program or data addresses,

just adjacent blobs. At any moment there is only *a single program cursor* and *a single data cursor*, connected by a bond written `*` above.

Instructions, in general. The blob instructions correspond roughly to "four-address code" for a von Neumann-style computer. An essential difference, though, is that a bond is a *two-way link between two blobs*, and is not an address at all. It is not a pointer; there *exists no address space* as in a conventional computer. A blob's 4 bond sites contain links to other instructions, or to data via the APB-ADB bond.

For program execution, one of the 8 cargo bits is an "activation bit"; if 1, it marks the instruction currently being executed. The remaining 7 cargo bits are interpreted as a 7-bit instruction so there are $2^7 = 128$ possible instructions in all. An instruction has an *operation code* (around 15 possibilities), and 0, 1 or 2 *parameters* that identify single bits, or bond sites, or cargo bits in a blob. See the table below for some current details. For example, `SCG v c` has 16 different versions since `v` can be one of 2 values, and `c` can be one of 8 values.

Why exactly 4 bonds? The reason is that each instruction must have a bond to its predecessor; further, a test or "jump" instruction will have two successor bonds (true and false); and finally, there must be one bond to link the APB and the ADB, i.e., the bond `*` between the currently executing instruction and the currently visible data blob. The `FIN` instruction is a device to allow a locally limited fan-in.

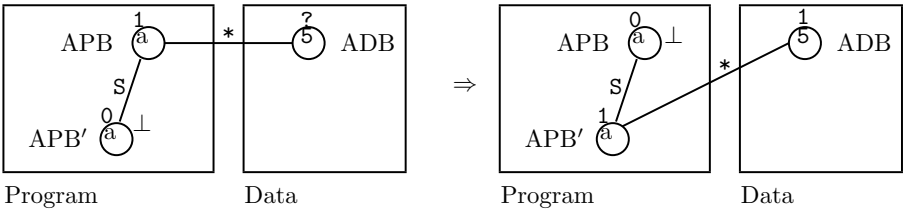
A specific instruction set (a bit arbitrary)**.** In [67,68] we propose a specific instruction set. Here we include a subset of the instructions for illustration.

On the insert instruction `INS`. This creates a new blob, linked with the current ADB. Analogy: thinking of a blob as a cell or molecule (whichever paradigm seems natural), we are implicitly assuming that blobs are swimming in a "biological soup", so `INS` just reconfigures a nearby element. From one viewpoint, this action resembles detaching a new cell from the freelist (the list of available cells used in Lisp/Scheme implementations).

Instruction	Description	Informal semantics
`SCG v c`	Set CarGo bit	ADB.c := v; APB := APB.2
`JCG c`	Jump CarGo bit	if ADB.c = 0 then APB := APB.3 else APB := APB.2
`JB b`	Jump Bond	if ADB.b = ⊥ then APB := APB.3 else APB := APB.2
`CHD b`	CHange Data	ADB := ADB.b; APB := APB.2
`INS b1 b2`	INSert new bond	new.b2 *bound to* ADB.b1; new.b1 *bound to* ADB.b1.bs; APB := APB.2 "new" is a fresh blob, "bs" is the bond site that ADB.b1 was bound to before `INS b1 b2`. ...
`FIN`	Fan IN	APB := APB.2

On the need for a fan-in instruction. The point with `FIN` (short for "fan-in") is that in blob code, unlike say Scheme, there cannot exist an unbounded number of pointers to a given blob (since every pointer corresponds to a bond site, and every blob has only 4 bond sites). So to achieve the effect of, say, 5 pointers to a blob instruction in a program, one can use a fan-in tree where each blob in the tree has at most 4 bond sites.

An example in detail: the instruction `SCG 1 5`, as picture and as a rewrite rule. SCG stands for "set cargo bit". The effect of instruction `SCG 1 5` is to change the 5-th cargo bit of the ADB (active data blob) to 1. First, an informal picture to show its effect:



Note: the APB-ADB bond `*` has moved: Before execution, it connected APB with ADB. After execution, it connects APB′ with ADB, where APB′ is the next instruction: the successor (via bond `S`) of the previous APB. Also note that the activation bit has changed: before, it was 1 at APB (indicating that the APB was about to be executed) and 0 at ADB′. Afterwards, those two bit values have been interchanged.

5.1 The Blob World from a Computer Science Perspective

First, an operational image: Any well-formed blob program, while running, is a collection of program blobs that is adjacent to a collection of data blobs, such that there is *one* critical bond (`*`) that links the APD and the ADB (the active program blob and the active data blob). As the computation proceeds, the program or data may move about, e.g., rotate as needed to keep their contact points adjacent (the APB and the ADB). For now, we shall not worry about the thermodynamic efficiency of moving arbitrarily large program and data in this way; for most realistic programs, we assume them to be sufficiently small (on the order of thousands of blobs) that energy considerations and blob coherence are not an issue.

5.2 The Blob Language

It is certainly small: around 15 operation codes (for a total of 128 instructions if parameters are included). Further, the set is irredundant in that no instruction's effect can be achieved by a combination of other instructions. There are easy computational tasks that simply cannot be performed by any program without, say, `SCG` or `FIN`.

There is a close analogy between blob programs and a *rudimentary machine language*. However a bond is not an address, but closer to a two-way pointer. On the other hand, there is *no address space*, and *no address decoding hardware* to move data to and from memory cells. An instruction has an unusual format, with 8 single bits and 4 two-way bonds. There is no fixed word size for data, there are no computed addresses, and there are no registers or indirection.

The blob programs have some similarity to *LISP or SCHEME*, but: there are no variables; there is no recursion; and bonds have a "fan-in" restriction.

5.3 What Can Be Done in the Blob World?

In principle the ideas presented and further directions are clearly expressible and testable in Maude or another tool for implementing term rewriting systems, or the kappa-calculus [26,31,40,44]. Recent work involves programming a blob simulator, and execution visualiser. Prototype implementations of both have been made, described in [67,68].

The usual programming tasks (appending two lists, copying, etc.) can be solved straightforwardly, albeit not very elegantly because of the low level of blob code.

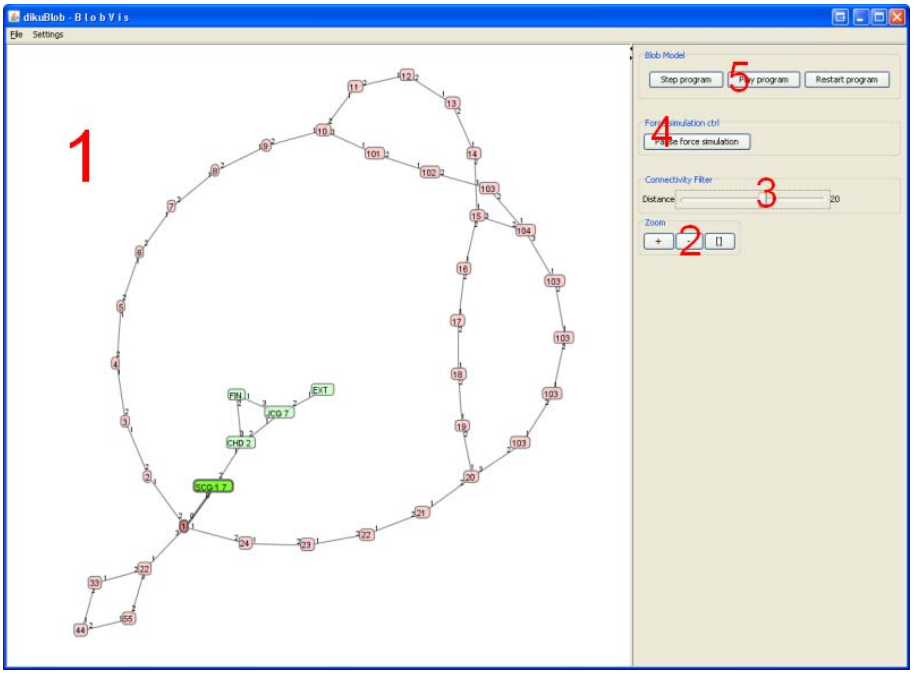


Fig. 8. The main interface of the blob visualiser. 1) visualisation area, 2) Zoom buttons, 3) Connectivity filter, 4) force-directed layout start/pause button, 5) blob program controls. Program blobs are green, data blobs red; the APB and ADB are emphasized by brighter colours and thicker bond lines.

It seems possible to make an analogy between universality and self-reproduction that is tighter than seen in the von Neumann and other cellular automaton approaches. It should now be clear that familiar Computer Science concepts such as interpreters and compilers also make sense also at the biological level, and hold the promise of becoming useful operational and utilitarian tools.

5.4 Blob Instruction Interpreter and Visualization Tool

We have developed an interpreter and visualization tool for the blob instruction set. Figure 8 shows an example of such an illustration. At `http://blobvis.appspot.com` the program and source code can be downloaded as well as several videos and images demonstrating programs and the usage of the visualization tool.

6 Conclusions

Computation via biological devices has been the subject of diverse and close scrutiny for many years. We have given a review of the literature on programming-related biocomputing and briefly identified some strengths and shortcomings from a programming perspective. Given the vast amount of work done in this field to date, the notion of programming the system or devices presented still seems to be ill-defined. Many models are claimed to be computationally universal in some sense. This universality prompted us to ask the question *Where are the programs?* Our recent work [67,68] tries to answer this question, where we presented the outline of a computation model that seems biologically more plausible than existing silicon-inspired models. We hope that it sets a standard for a biological device that can be both universal *and programmable*.

References

1. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science 266(11), 1021–1024 (1994)
2. Adleman, L.M.: On constructing a molecular computer. DIMACS: series in Discrete Mathematics and Theoretical Computer Science, pp. 1–21. American Mathematical Society (1996)
3. Amos, M., Paun, G., Rozenberg, G., Salomaa, A.: Topics in the theory of DNA computing. Theor. Comput. Sci. 287(1), 3–38 (2002)
4. Backofen, R., Clote, P.: Evolution as a computational engine. In: Proceedings of the Annual Conference of the European Association for Computer Science Logic, pp. 35–55. Springer, Heidelberg (1996)
5. Beaver, D.: Computing with dna. Journal of Computational Biology 2(1), 1–7 (1995)
6. Beaver, D.: Computing with DNA. Journal of Computational Biology 2(1), 1–7 (1995)
7. Benenson, Y., Adar, R., Paz-Elizur, T., Livneh, Z., Shapiro, E.: Dna molecule provides a computing machine with both data and fuel. Proc Natl Acad Sci U S A 100(5), 2191–2196 (2003), `http://dx.doi.org/10.1073/pnas.0535624100`

8. Benenson, Y., Adar, R., Paz-Elizur, T., Livneh, Z., Shapiro, E.: DNA molecule provides a computing machine with both data and fuel. In: Noltemeier, H. (ed.) WG 1980. LNCS, vol. 100, pp. 2191–2196. Springer, Heidelberg (1981)
9. Benenson, Y., Paz-Elizur, T., Adar, R., Keinan, E., Livneh, Z., Shapiro, E.: Programmable and autonomous computing machine made of biomolecules. Nature 414(1), 430–434 (2001), http://www.nature.com/nature/links/011122/011122-2.html
10. Benenson, Y.: Biocomputers: from test tubes to live cells. Molecular BioSystems 5(7), 675–685 (2009), http://dx.doi.org/10.1039/b902484k
11. Benenson, Y.: RNA-based computation in live cells. Current Opinion in Biotechnology 20(4), 471–478 (2009), http://www.sciencedirect.com/science/article/B6VRV-4X4BR27-2/2/0133dc1fc3a23441b6aa9bab4115fc11, protein technologies / Systems and synthetic biology
12. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development 17(6), 525–532 (1973)
13. Bennett, C.H.: The thermodynamics of computation – a review. International Journal of Theoretical Physics 21(12), 905–940 (1982), http://dx.doi.org/10.1007/BF02084158
14. Berger, R.: The undecidability of the domino problem. Memoirs American Mathematical Society 66 (1966)
15. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning Ways for your Mathematical Plays, vol. 2, ch. 25. Academic Press (1982) ISBN 0-12-091152-3
16. Bohringer, K.-F., Paulisch, N.F.: Using constraints to achieve stability in automatic graph layout algorithms. In: Proceedings of ACM CHI 1990 Conference on Human Factors in Computing Systems, pp. 43–51. Constraint Based UI Tools (1990)
17. Boneh, D., Dunworth, C., Lipton, R.J.: Breaking DES using a molecular computer. In: Lipton, E.B.B.R.J. (ed.) DNA Based Computers. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 27, pp. 37–66. American Mathematical Society (1995)
18. Braich, R.S., Chelyapov, N., Johnson, C., Rothemund, P.W.K., Adleman, L.: Solution of a 20-variable 3-sat problem on a dna computer. Science 296, 499–502 (2002)
19. Cardelli, P.: An universality result for a (mem)brane calculus based on mate/drip operations. IJFCS: International Journal of Foundations of Computer Science 17 (2006)
20. Cardelli, L.: Brane calculi. In: Danos and Schächter [14], pp. 257–278, http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3082&spage=257
21. Cardelli, L.: Abstract machines of systems biology. In: Priami, C., Merelli, E., Gonzalez, P., Omicini, A. (eds.) Transactions on Computational Systems Biology III. LNCS (LNBI), vol. 3737, pp. 145–168. Springer, Heidelberg (2005)
22. Cardelli, L.: Strand algebras for DNA computing. In: Deaton and Suyama [42], pp. 12–24
23. Cardelli, L.: Molecular programming tutorial, microsoft research, cambridge (February 2010), http://lucacardelli.name/Talks/2010-02-11%20Molecular%20Programming%20Tutorial.pdf
24. Cardelli, L.: Biocomputers is not a good idea of solving np complete problems. said during presentation of strand algebra, CS2Bio, Amsterdam (2010)

25. Cardelli, L., Zavattaro, G.: On the computational power of biochemistry. In: Horimoto, K., Regensburger, G., Rosenkranz, M., Yoshida, H. (eds.) AB 2008. LNCS, vol. 5147, pp. 65–80. Springer, Heidelberg (2008)
26. Cardelli, L., Zavattaro, G.: Turing universality of the biochemical ground form. Mathematical Structures in Computer Science 19 (2009)
27. Chang, W.L., Ho, M.H., Guo, M.: Molecular solutions for the subset-sum problem on DNA-based supercomputing. Biosystems 73, 117–130(14) (2004), http://www.ingentaconnect.com/content/els/03032647/2004/00000073/00000002/art00225
28. Chapman, P.: Life universal computer (November 2002), http://www.igblan.free-online.co.uk/igblan/ca/
29. Chiniforooshan, E., Doty, D., Kari, L., Seki, S.: Scalable, time-responsive, digital, energy-efficient molecular circuits using DNA strand displacement. CoRR abs/1003.3275 (2010)
30. Ciocchetta, F., Hillston, J.: Bio-PEPA: An extension of the process algebra PEPA for biochemical networks. Electr. Notes Theor. Comput. Sci. 194(3), 103–117 (2008), http://dx.doi.org/10.1016/j.entcs.2007.12.008
31. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
32. Condon, A., Rozenberg, G. (eds.): DNA 2000. LNCS, vol. 2054. Springer, Heidelberg (2001)
33. Conrad, M.: On design principles for a molecular computer. Commun. ACM 28(5), 464–480 (1985)
34. Conrad, M.: Quantum molecular computing: The self-assembly model. International Journal of Quantum Chemistry. Quantum Biology Symposium: Proceedings of the International Symposium on Quantum Biology and Quantum Pharmacology, vol. 19, pp. 125–143 (1992)
35. Danchin, A.: Bacteria as computers making computers. FEMS Microbiology Reviews 33(1), 3–26 (2008)
36. Danos, V., Feret, J., Fontana, W., Krivine, J.: Abstract interpretation of cellular signalling networks. In: Logozzo, et al [83], pp. 83–97
37. Danos, V., Feret, J., Fontana, W., Krivine, J.: Abstract interpretation of cellular signalling networks. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 83–97. Springer, Heidelberg (2008)
38. Danos, V., Krivine, J.: Formal molecular biology done in CCS-R. Electr. Notes Theor. Comput. Sci. 180(3), 31–49 (2007), http://dx.doi.org/10.1016/j.entcs.2004.01.040
39. Danos, V., Laneve, C.: Formal molecular biology. Theor. Comput. Sci. 325(1), 69–110 (2004)
40. Danos, V., Laneve, C.: Formal molecular biology. Theor. Comp. Science 325, 69–110 (2004)
41. Danos, V., Schachter, V. (eds.): CMSB 2004. LNCS (LNBI), vol. 3082. Springer, Heidelberg (2005)
42. Deaton, R., Suyama, A. (eds.): DNA 15. LNCS, vol. 5877. Springer, Heidelberg (2009)
43. Degano, P., Gorrieri, R. (eds.): CMSB 2009. LNCS, vol. 5688. Springer, Heidelberg (2009)

44. Delzanno, G., Giusto, C.D., Gabbrielli, M., Laneve, C., Zavattaro, G.: The *kappa*-lattice: Decidability boundaries for qualitative analysis in biological languages. In: Degano and Gorrieri [43], pp. 158–172
45. Doty, D., Patitz, M.J.: A Domain-Specific Language for Programming in the Tile Assembly Model, pp. 25–34. Springer, Heidelberg (2009)
46. Eades, P.: A heuristic for graph drawing. Congressus Numerantium 42, 149–160 (1984)
47. Eades, P., Lai, W., Misue, K., Sugiyama, K.: Preserving the mental map of a diagram. In: COMPUGRAPHICS 1991, vol. I, pp. 34–43 (1991)
48. Elbaz, J., Lioubashevski, O., Wang, F., Remacle, F., Levine, R.D., Willner, I.: DNA computing circuits using libraries of DNAzyme subunits. Nat. Nanotechnol. 5(6), 417–422 (2010), http://dx.doi.org/10.1038/nnano.2010.88
49. Fisher, J., Henzinger, T.A.: Executable cell biology. Nature Biotechnology 25(11), 1239–1249 (2007), http://dx.doi.org/10.1038/nbt1356
50. Fleischer, R., Hirsch, C.: Graph drawing and its applications. In: Kaufmann, M., Wagner, D. (eds.) Drawing Graphs. LNCS, vol. 2025, pp. 1–22. Springer, Heidelberg (2001)
51. Fontana, W., Buss, L.: The barrier of objects: From dynamical systems to bounded organizations. Working Papers wp96027, International Institute for Applied Systems Analysis (March 1996), http://ideas.repec.org/p/wop/iasawp/wp96027.html
52. Frick, A., Ludwig, A., Mehldau, H.: A fast adaptive layout algorithm for undirected graphs. In: Tamassia, R., Tollis, I.G. (eds.) GD 1994. LNCS, vol. 894, pp. 388–403. Springer, Heidelberg (1995), http://dblp.uni-trier.de/db/conf/gd/gd94.html#FrickLM94
53. Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. Software: Practice and Experience 21(11), 1129–1164 (1991), citeseer.ist.psu.edu/fruchterman91graph.html
54. Fu, P.: Biomolecular computing: Is it ready to take off? Biotechnology Journal 2(1), 91–101 (2007), http://dx.doi.org/10.1002/biot.200600134
55. Gardner, M.: The fantastic combinations of John Conway's new solitaire game "life". Scientific American 223, 120–123 (1970)
56. Gardner, M.: Mathematical recreations. Scientific American (October 1970)
57. Garzon, M.H., Deaton, R.J.: Biomolecular computing and programming. IEEE Trans. Evolutionary Computation 3(3), 236–250 (1999)
58. Gilmore, S., Hillston, J.: The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In: Haring, G., Kotsis, G. (eds.) TOOLS 1994. LNCS, vol. 794, pp. 353–368. Springer, Heidelberg (1994)
59. Giral, U.D.E., Cetintas, A., Civril, A., Demir, E.: A compound graph layout algorithm for biological pathways. In: Pach [94], pp. 442–447, http://dblp.uni-trier.de/db/conf/gd/gd2004.html#DogrusozGCCD04
60. Goel, A., Ibrahimi, M.: Renewable, time-responsive DNA logic gates for scalable digital circuits. In: Deaton and Suyama [42], pp. 67–77
61. Goel, A., Simmel, F.C., Sosík, P. (eds.): DNA Computing. LNCS, vol. 5347. Springer, Heidelberg (2009)
62. Guarnieri, F., Fliss, M., Bancroft, C.: Making DNA add. Science 273(5272), 220–223 (1996)

63. Guerriero, M.L., Prandi, D., Priami, C., Quaglia, P.: Process calculi abstractions for biology. Tech. rep., CoSBi (Center for Computational and Systems Biology), University of Trento (January 01, 2006), http://eprints.biblio.unitn.it/archive/00001704/, http://eprints.biblio.unitn.it/archive/00001704/01/TR-13-2006.pdf
64. Guerriero, M.L., Prandi, D., Priami, C., Quaglia, P.: Process calculi abstractions for biology. Tech. rep., University of Trento, Italy (January 01, 2006), http://eprints.biblio.unitn.it/archive/00001704/, http://eprints.biblio.unitn.it/archive/00001704/01/TR-13-2006.pdf
65. Hagiya, M.: From molecular computing to molecular programming. In: Condon and Rozenberg [32], pp. 89–102, http://link.springer.de/link/service/series/0558/bibs/2054/20540089.htm
66. Hagiya, M.: Designing chemical and biological systems. New Generation Comput. 26(3), 295 (2008)
67. Hartmann, L., Jones, N., Simonsen, J.: Programming in biomolecular computation. In: CS2BIO 2009: Proceedings of the 1st International Workshop on Interactions between Computer Science and Biology. Electronic Notes on Theoretical Computer Science series. Elsevier (2010), http://dx.doi.org/10.1016/j.entcs.2010.12.008
68. Hartmann, L., Jones, N., Simonsen, J., Vrist, S.: Programming in biomolecular computation: Programs, self-interpretation and visualisation. To appear in Scientific Annals of Computer Science, http://dk.diku.blob.blobvis.s3.amazonaws.com/blobiasi.pdf
69. Heer, J.: Prefuse: a software framework for interactive information visualization. Master's thesis, University of California, Berkeley (2004), http://jheer.org/publications/2004-Heer-prefuse-MastersApp.pdf
70. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
71. Hug, H., Schuler, R.: Strategies for the development of a peptide computer. Bioinformatics 17(4), 364–368 (2001), http://bioinformatics.oxfordjournals.org/content/17/4/364.abstract
72. Jones, J.E.: On the determination of molecular fields. ii. from the equation of state of a gas. Proceedings of the Royal Society of London. Series A 106(738), 463–477 (1924), http://dx.doi.org/10.1098/rspa.1924.0082
73. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall International Series in Computer Science. Prentice-Hall (1993)
74. Jones, N.D.: Computability and complexity: from a programming perspective. MIT Press, Cambridge (1997)
75. Kamada, T., Kawai, S.: An algorithm for drawing general undirected graphs. Information Processing Letters 31(1), 7–15 (1989)
76. Kari, J.: Theory of cellular automata: A survey. Theoretical Computer Science 334(1-3), 3–33 (2005), http://www.sciencedirect.com/science/article/B6V1G-4FDS8HM-2/2/7bdf589f505353432c8447e06f491ceb
77. Kari, L.: Biological computation: How does nature compute? Tech. rep., University of Western Ontario (2009)
78. Kari, L., Rozenberg, G.: The many facets of natural computing. Commun. ACM 51(10), 72–83 (2008)
79. Khodor, J.: DNA-based string rewrite computational systems. Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science (2002), http://hdl.handle.net/1721.1/8339

80. Khodor, J., Gifford, D.K.: Programmed mutagenesis is universal. Theory Comput. Syst. 35(5), 483–500 (2002), http://dblp.uni-trier.de/db/journals/mst/mst35.html#KhodorG02
81. Krämer, M., Pita, M., Zhou, J., Ornatska, M., Poghossian, A., Schöning, M.J., Katz, E.: Coupling of biocomputing systems with electronic chips: Electronic interface for transduction of biochemical information. The Journal of Physical Chemistry C 113(6), 2573–2579 (2009), http://pubs.acs.org/doi/abs/10.1021/jp808320s
82. Lipton, R.J.: Using DNA to solve NP-complete problems. Science 268, 542–545 (1995)
83. Logozzo, F., Peled, D., Zuck, L.D. (eds.): VMCAI 2008. LNCS, vol. 4905. Springer, Heidelberg (2008)
84. Lund, K., Manzo, A.J., Dabby, N., Michelotti, N., Johnson-Buck, A., Nangreave, J., Taylor, S., Pei, R., Stojanovic, M.N., Walter, N.G., Winfree, E., Yan, H.: Molecular robots guided by prescriptive landscapes. Nature 465(7295), 206–210 (2010), http://dx.doi.org/10.1038/nature09012
85. Mao, C., Labean, T.H., Reif, J.H., Seeman, N.C.: Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. Nature 407, 493–496 (2000)
86. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
87. Milner, R.: Functions as processes. Research Report 1154, INRIA (1990)
88. Minsky, M.: Computation: finite and infinite machines. Prentice-Hall, Englewood Cliffs (1967)
89. Misue, K., Eades, P., Lai, W., Sugiyama, K.: Layout adjustment and the mental map. Journal of Visual Languages and Computing 6(2), 183–210 (1995), http://www.sciencedirect.com/science/article/B6WMM-45PVMS3-13/2/0f1f0f6cf4f49a7892fb6064751b128c
90. Stojanovic, M.N., Stefanovic, D.: A deoxyribozyme-based molecular automaton. Nature Biotechnol. 21(9), 1069–1074 (2003)
91. Murata, S., Stojanovic, M.N.: DNA-based nanosystems. New Generation Comput. 26(3), 297–312 (2008)
92. Nehaniv, C.L.: Asynchronous automata networks can emulate any synchronous automata network. International Journal of Algebra and Computation 14(5-6), 719–739 (2004)
93. von Neumann, J., Burks, A.W.: Theory of Self-Reproducing Automata. Univ. Illinois Press (1966)
94. Pach, J. (ed.): GD 2004. LNCS, vol. 3383. Springer, Heidelberg (2005)
95. Parker, J.: Computing with DNA. EMBO Rep. 4(7), 7–10 (2003)
96. Phillips, A., Cardelli, L.: A programming language for composable DNA circuits. Journal of the Royal Society Interface 6(S4) (2009)
97. Qian, L., Winfree, E.: A simple DNA gate motif for synthesizing large-scale circuits. In: Goel, et al [61], pp. 70–89
98. Ran, T., Kaplan, S., Shapiro, E.: Molecular implementation of simple logic programs. Nat. Nano. 4(10), 642–648 (2009), http://dx.doi.org/10.1038/nnano.2009.203
99. Regev, A., Shapiro, E.Y.: Cells as computation. In: Priami, C. (ed.) CMSB 2003. LNCS, vol. 2602, pp. 1–3. Springer, Heidelberg (2003), http://link.springer.de/link/service/series/0558/bibs/2602/26020001.htm

100. Regev, A., Panina, E.M., Silverman, W., Cardelli, L., Shapiro, E.: Bioambients: An abstraction for biological compartments. TCS: Theoretical Computer Science 325 (2004)
101. Regev, A., Silverman, W., Shapiro, E.Y.: Representation and simulation of biochemical processes using the pi-calculus process algebra. In: Pacific Symposium on Biocomputing, pp. 459–470 (2001), http://helix-web.stanford.edu/psb01/regev.pdf
102. Reif, J.H., LaBean, T.H.: Autonomous programmable biomolecular devices using self-assembled DNA nanostructures. Commun. ACM 50(9), 46–53 (2007), http://dblp.uni-trier.de/db/journals/cacm/cacm50.html#ReifL07
103. Robinson, R.M.: Undecidability and nonperiodicity for tilings of the plane. Inv. Math. 12, 117–209 (1971)
104. Rothemund, P.W.K.: A DNA and restriction enzyme implementation of Turing machines. In: Lipton, E.B.B.R.J. (ed.) DNA Based Computers. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 27, pp. 75–120. American Mathematical Society (1995)
105. Rothemund, P.W.K.: Using lateral capillary forces to compute by self-assembly. Proceedings of the National Academy of Sciences of the United States of America 97(3), 984–989 (2000), http://www.pnas.org/content/97/3/984.abstract
106. Rothemund, P.: Folding DNA to create nanoscale shapes and patterns. Nature 440, 297–302 (2006)
107. Roweis, S., Winfree, E., Burgoyne, R., Chelyapov, N.V., Goodman, M.F., Rothemund, P.W.K., Adleman, L.M.: A sticker based model for DNA computation. In: Landweber, L., Baum, E. (eds.) DNA Based Computers II. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 44, American Mathematical Society (1996), ftp://hope.caltech.edu/pub/roweis/DIMACS/stickers.ps
108. Roweis, S.T., Winfree, E., Burgoyne, R., Chelyapov, N.V., Goodman, M.F., Rothemund, P.W.K., Adleman, L.M.: A sticker-based model for DNA computation. Journal of Computational Biology 5(4), 615–630 (1998)
109. Sander, G.: Graph layout for applications in compiler construction. Theor. Comput. Sci. 217(2), 175–214 (1999), http://dblp.uni-trier.de/db/journals/tcs/tcs217.html#Sander99
110. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press (2001)
111. Seelig, G., Soloveichik, D.: Time-Complexity of Multilayered DNA Strand Displacement Circuits, pp. 144–153. Springer, Heidelberg (2009)
112. Seelig, G., Soloveichik, D., Zhang, D.Y., Winfree, E.: Enzyme-Free Nucleic Acid Logic Circuits. Science 314(5805), 1585–1588 (2006), http://www.sciencemag.org/cgi/content/abstract/314/5805/1585
113. Shapiro, B.: Bringing DNA computers to life. SCIAM: Scientific American 294 (2006)
114. Shapiro, E.: Mechanical Turing machine: Blueprint for a biomolecular computer. Tech. rep., Weizmann Institute of Science (1999)
115. Shapiro, E.: Mechanical Turing machine: Blueprint for a biomolecular computer. Tech. rep., Weizmann Institute of Science (1999)
116. Shapiro, E., Benenson, Y.: Bringing DNA computers to life. Scientific American 294, 44–51 (2006)
117. Shlyahovsky, B., Li, Y., Lioubashevski, O., Elbaz, J., Willner, I.: Logic gates and antisense DNA devices operating on a translator nucleic acid scaffold. ACS Nano 3(7), 1831–1843 (2009), http://dx.doi.org/10.1021/nn900085x

118. Simpson, M.L., Sayler, G.S., Fleming, J.T., Applegate, B.: Whole-cell biocomputing. Trends Biotechnol. 19(8), 317–323 (2001), http://www.biomedsearch.com/nih/Whole-cell-biocomputing/11451474.html
119. Smith, W.D.: DNA computers in vitro and vivo. In: Lipton, E.B.B.R.J. (ed.) DNA Based Computers. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 27, pp. 121–186. American Mathematical Society (1995)
120. Soloveichik, D., Seelig, G., Winfree, E.: DNA as a universal substrate for chemical kinetics. In: Goel et al [61], pp. 57–69
121. Stefansen, C.: SMAWL: A SMAll workflow language based on CCS. In: Belo, O., Eder, J. (eds.) CAiSE 2005. CAiSE Forum, Short Paper Proceedings. CEUR Workshop Proceedings, vol. 161, CEUR-WS.org (2005), http://www.ceur-ws.org/Vol-161/FORUM_10.pdf
122. Storey, M.A.D., Fracchia, F., Müller, H.: Customizing a Fisheye View Algorithm to Preserve the Mental Map. Journal of Visual Languages and Computing 10(3), 245–267 (1999)
123. Talcott, C.: Pathway logic. In: Bernardo, M., Degano, P., Tennenholtz, M. (eds.) SFM 2008. LNCS, vol. 5016, pp. 21–53. Springer, Heidelberg (2008)
124. Turing, A.: On computable numbers with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 42(2), 230–265 (1936-1937)
125. Wang, H.: Proving theorems by pattern recognition ii. Bell System Technical Journal 40, 1–40 (1961)
126. Wang, S., Yang, A.: DNA solution of integer linear programming. Applied Mathematics and Computation 170(1), 626–632 (2005)
127. Winfree, E.: Toward molecular programming with DNA. SIGOPS Oper. Syst. Rev. 42(2), 1–1 (2008)
128. Winfree, E., Eng, T., Rozenberg, G.: String tile models for DNA computing by self-assembly. In: Condon, A., Rozenberg, G. (eds.) DNA 2000. LNCS, vol. 2054, pp. 63–88. Springer, Heidelberg (2001)
129. Winfree, E., Yang, X., Seeman, N.C.: Universal computation via self-assembly of DNA: Some theory and experiments. In: DNA Based Computers II. DIMACS, vol. 44, pp. 191–213. American Mathematical Society (1996)
130. Winfree, E., Yang, X., Seeman, N.C.: Universal computation via self-assembly of DNA: Some theory and experiments. In: DNA Based Computers II. DIMACS, vol. 44, pp. 191–213. American Mathematical Society (1996)
131. Wolfram, S.: A New Kind of Science. Wolfram Media (January 2002), http://www.amazon.com/exec/obidos/ASIN/1579550088/ref=nosim/rds-20
132. Yan, H., Park, S.H., Finkelstein, G., Reif, J.H., Labean, T.H.: DNA-templated self-assembly of protein arrays and highly conductive nanowires. Science 301(5641), 1882–1884 (2003), http://dx.doi.org/10.1126/science.1089389
133. Yin, P., Choi, H.M.T., Calvert, C.R., Pierce, N.A.: Programming biomolecular self-assembly pathways. Nature 451(7176), 318–322 (2008), http://dx.doi.org/10.1038/nature06451
134. Yin, P., Turberfield, A.J., Sahu, S., Reif, J.H.: Design of an autonomous DNA nanomechanical device capable of universal computation and universal translational motion. In: Ferretti, C., Mauri, G., Zandron, C. (eds.) DNA 2004. LNCS, vol. 3384, pp. 426–444. Springer, Heidelberg (2005)

135. Yokomori, T., Kobayashi, S., Ferretti, C.: On the power of circular splicing systems and DNA computability. In: IEEE International Conference on Evolutionary Computation (1997), http://ylab-gw.cs.uec.ac.jp/../Papers/yokomori/cssfinal.ps.gz
136. Zhang, D.Y.: Dynamic DNA strand displacement circuits. Ph.D. thesis, California Institute of Technology (2010), http://resolver.caltech.edu/CaltechTHESIS:05262010-173410602
137. Zhu, Y., Ding, Y., Li, W., Kemp, G.: A proposed modularized dna computer, based on biochips. In: GEC 2009: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation, pp. 773–780. ACM, New York (2009)