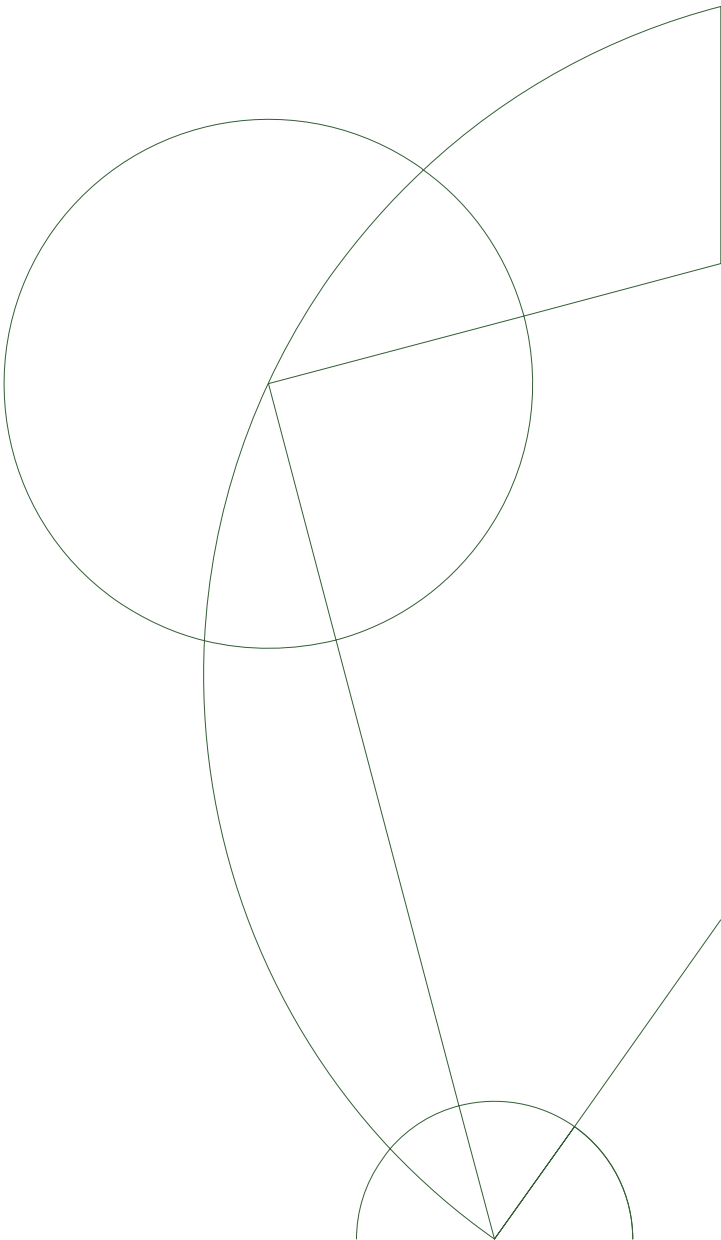
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Niels Bjørn Bugge Grathwohl

The Biological Plausibility of the Blob Model



Supervisor: Jakob Grue Simonsen

Department of Computer Science, September 2011

Abstract

A biomolecular computer is a computer constructed using materials and concepts borrowed from the disciplines biochemistry and molecular biology. So far, biomolecular computers have been more akin to specialized Boolean circuits than to a "computer" in the normal understanding of the word, in the sense that each computer is constructed for one specific purpose, thus not being programmable.

The Blob Model is an abstract computational model that models a "biologically plausible," naturally programmable computer. This thesis concerns the actual realizability in biomolecular substrates of the Blob Model. We present the following results: a) a survey of existing work on biomolecular computation; b) a classification of computers into either of the two groups *mechanically universal* and *linguistically universal*, establishing a distinction between Turing-universality in the sense that any computable function can be *built* and Turing-universality in the sense that any computable function can be *programmed*; c) an evaluation of the realizability of the Blob Model in the context of DNA self-assembly, four different versions of biomolecular Boolean gates, and *FokI* restriction; d) a theoretical implementation scheme, disregarding the use of actual tested laboratory methods.

We find that the Blob Model cannot be realized in any one of the studied biomolecular substrates. This indicates that the Blob Model should be revised in order to maintain its biological plausibility.

Resumé

En biomolekylær computer er en computer der er konstrueret ved brug af materialer og koncepter lånt fra disciplinerne biokemi og molekylærbiologi. Indtil videre har biomolekylære computere mindet mere om specialiserede boolske kredsløb end om en egentlig "computer" i den dagligdags opfattelse af ordet, forstået således at hver enkelt computer er konstrueret til ét specifikt formål, hvorfor den ikke er programmerbar.

Blobmodellen er en abstrakt beregningsmodel som modellerer en "biologisk plausibel", naturligt programmerbar computer. Dette speciale omhandler realiserbarheden i en biomolekylær kontekst af Blobmodellen. Vi præsenterer følgende resultater: a) en undersøgelse af eksisterende arbejde på biomolekylære computere; b) en klassifikation af computere i én af de to grupper af *mekanisk universelle* og *lingvistisk universelle* computere, for dermed at etablere en skelnen mellem Turing-fuldstændighed i den forstand at alle beregnelige funktioner kan *bygges*, og Turing-fuldstændighed i den forstand at alle beregnelige funktioner kan *programmeres*; c) en evaluering af realiserbarheden af Blobmodellen ved brug af selvsamlende DNA, fire forskellige versioner af biomolekylære boolske gates, samt *FokI*-restriktion; d) et forslag til en teoretisk fremgangsmåde for en implementering, formuleret uden hensyntagen til nødvendigheden af brugen af egentlige, laboratorielt afprøvede metoder.

Vi viser, at Blobmodellen ikke kan realiseres i nogen af de studerede biomolekylære substrater. Dette indikerer at Blobmodellen skal revideres, hvis den skal bevare sin biologiske plausibilitet.

Contents

List of Figures

Preface

This is a thesis for the degree of *cand.scient*, or Master of Science, at the University of Copenhagen, Department of Computer Science.

Prior to writing this text, I had no experience with biology. Part of the excitement in doing this work stems from the revelation of the tiny subset of molecular biology that I have been subject to.

Acknowledgements. My supervisor, *Jakob Grue Simonsen*, has been of enormous help as a discussion partner, as an unquenchable optimist, and as a tireless slave driver. I have also received much from *Christine Søholm Hansen* and *Nicolai Skovbjerg Arildsen*, as they have patiently corrected the several misconceptions of biological concepts formed in my head during work on this thesis. *Hans*, my brother the mathematician, has provided ample, solid, and necessary support on a lot of the formalities. Finally, my girlfriend *Christine* has been supportive and loving, without which nothing could ever be done.

— Niels Bjørn Bugge Grathwohl
Copenhagen, 2011

Introduction

The aim of this thesis is to answer the following question: *Can the* blob model*, introduced by Hartmann, Jones, and Simonsen in [54], be realized using existing biomolecular techniques that are used in the construction of other biomolecular computing models?*

❧

Does programmability pertain to the computers constructed in the past 200 years only, or is the traditional notion of programmability merely one shade of a more fundamental property? Comparing the inner workings of a eukaryotic cell to those of a 2011-era PC, we certainly see some huge differences: The cell performs many tasks in an inherently parallel way, and the environment in which the cell operates is more "noisy" than that of the PC. Consequently, the notion of programming in a biological context is likely to be a lot different from what we know, if it is at all possible. In spite of this, attempts at unifying the rather orthogonal fields of biology and computer science are plentiful: enter biomolecular computing.

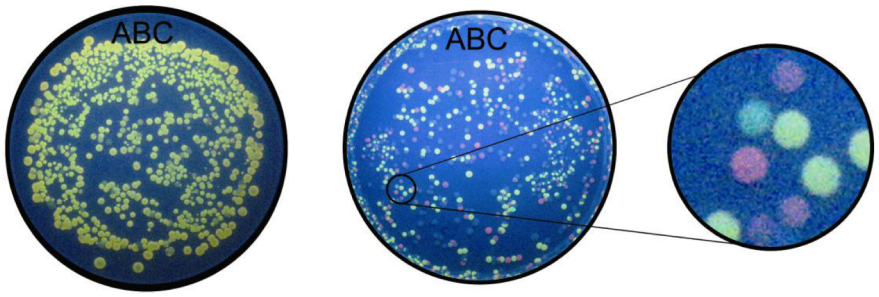


Figure 1: *Colonies of* E. coli*, demonstrating an expression of the solution to an instance of small Hamiltonian path problem. The bacteria that contain a representation of a correct solution fluoresce yellow.*

The term "biomolecular computing" covers a variety of goals and approaches. As noted, one of the properties of biological systems that sets them apart from classic computers is their inherent parallelism. Due to this observation, an early aspiration in the development of biomolecular computers were to find a possible solution route to large instances of NP-complete problems, thereby offering performance that traditional computers were far from achieving. This initial excitement has faded, however, as it has become evident that in spite of inherent parallelism, the biomolecular computers are unable to scale as the size of the problem instances grow larger.

Although biomolecular computers cannot solve large instances of NP-complete problems fast (the earliest breakthrough from 1994, solving the Hamiltonian path problem for a seven-node graph, is still comparable in size to the state of the art), they are still interesting. One should view the several efforts for constructing biomolecular computers not as an attempt to produce rivals to the existing electronic computers, but rather as an endeavor to bring the techniques,

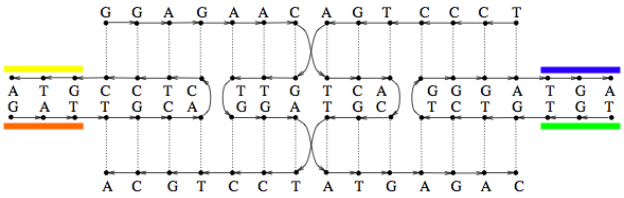


Figure 2: *The ability of DNA to self-assemble in a consistent way can is exploited by crafting special "tiles" of DNA whose formation can be regarded as a computer.*

approaches, and knowledge developed in the field of computer science to use in new areas: Sub-cellular-sized biomolecular computers acting directly on biological material may be capable of doing things that a modern supercomputer would be completely unable to do, even though the supercomputer very likely would be magnitudes faster than the biological computer when it comes to "raw" computing power. When blue-sky-dreaming, one sees medical uses due to the expected "natural interface" to biological objects, allowing scientists to program medicine in a much more systematic way which (at least from the point of view of a computer scientist) is more desirable than a less systematic trial-and-error-based methodology.

Still, in spite of great expectations, implementing computers with biomolecular techniques is still in its infancy. Interesting proof-of-concept implementations have been constructed, capable of performing basic operations such as logic functions or transitions in a deterministic finite automaton, but so far the demonstrated computers have been more akin to small, specialized circuits than what is commonly understood with a computer. As a result they are non-programmable, a property that we would normally deem to be a key component of a computer.

This observation was the motivation behind the work of Hartmann et al. when they developed the blob model; an abstract computational model designed with the objective that it should be "naturally programmable" and be realizable in some form of biomolecular substrate. The challenge in defining such a

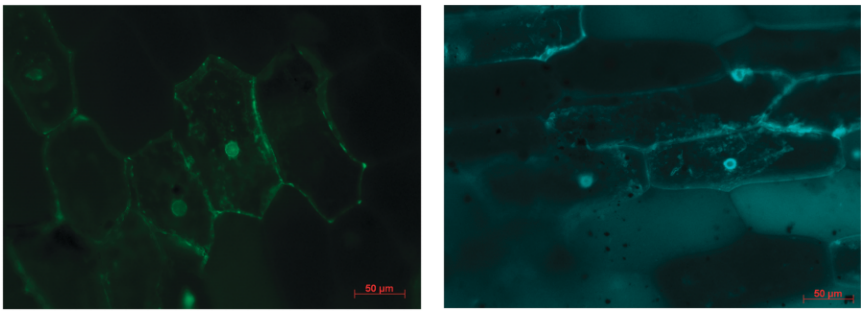


Figure 3: *Using green and cyan fluorescent protein injected into onion cells, it is revealed whether the final state of a DFA is accepting or rejecting.*

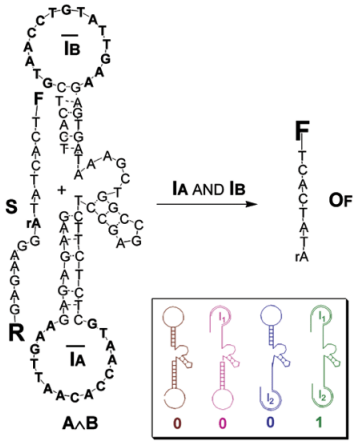


Figure 4: *A biomolecular AND gate, implemented with deoxyribozymes, a DNA structure with catalytic properties.*

model is that programmability needs to be conceptualized in the fuzzy world of "wetware," where the environment and the factors that have to be taken into account are drastically different from those of normal computers.

Equipped with this model, one "only" needs to find a suitable implementation substrate and technique, as the Turing completeness and programmability of the blob model have both been established theoretically. Looking at the approaches taken in previous work in biomolecular computing leaves an impression that this model is a very novel approach; the directionality of the design has been the opposite of many others, moving from a solid theoretical base towards the wet world of a laboratory instead of the other way around. Indeed, some authors simply ignore theoretical questions regarding the computer science behind their construction, although to be fair, those authors are not computer scientists themselves. This exemplifies the variation in goals found in the

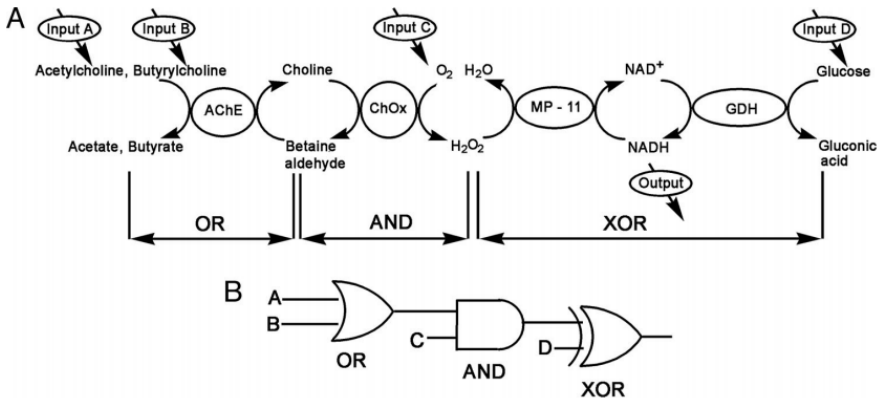


Figure 5: *An enzyme-based circuit, using the output of the catalyzed reactions of some enzymes as reactants in the reactions catalyzed by other enzymes.*

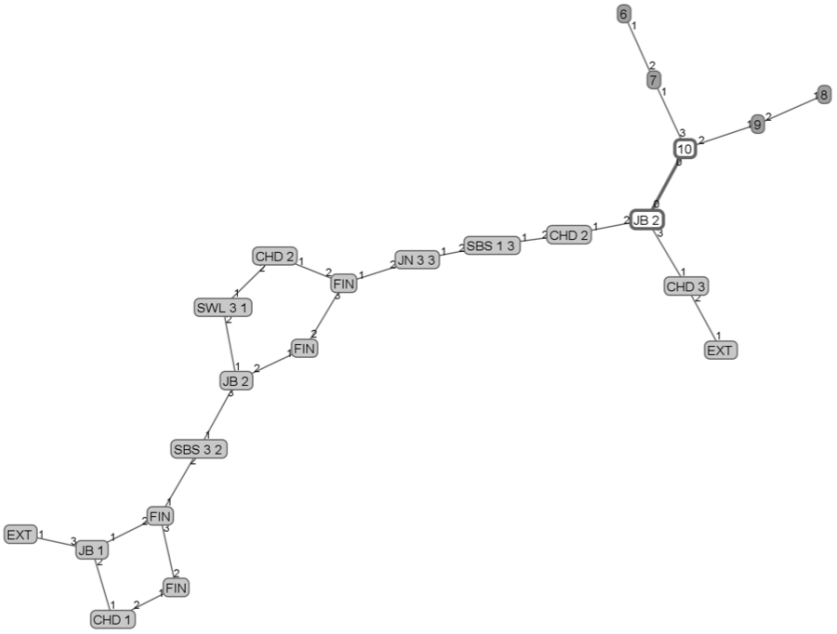


Figure 6: *A blob program's explicit physical nature causes data and program code to be representable as physically adjacent "blobs."*

research on biomolecular computers: Some are biochemists looking for clever ways to implement molecular switches, to which end they consider basic Boolean logic. Others are geneticists researching ways to control gene expression, while yet others are computer scientists hoping for a new, fast way to solve hard problems.

The general approach taken in this thesis is to conduct a broad literature survey and extract the methods that look most promising. We also provide a formal treatment of the notion "natural programmability" in order to have a firmer base when evaluating selected biomolecular techniques. To underline the generality of the notion "computer," electronic, biological, and mechanical examples are used in the treatment.

Five different techniques, all of them tested in proof-of-concept laboratory implementations of simple (non-programmable) computers, are evaluated. When used alone, none of them can be used as an implementation substrate for the blob model. When used in concert with each other, they still do not stand as implementation substrates unless some serious obstacles were addressed. Moving on from this saddening truth, we develop a more free-flowing implementation scheme, in which some liberties are taken as to what has actually been built in a laboratory. The hope with this latter part is to shed light on ways for future research in programmable biomolecular computers.

The ideal reader has a computer science education equivalent to that described by ACM's guidelines [1], along with introductory knowledge about computability theory roughly equivalent to the contents in [103]. Proficiencies in areas related to molecular biology is not necessary, as the concepts required to understand the points delineated in this thesis are introduced.

1 Preliminaries

Within this chapter the necessary basic concepts of molecular biology is introduced, along with a reminder of the concepts from theoretical computer science that are used in this thesis.

Readers familiar with the contents of Sipser's introduction to computability theory [103] or Alberts et al.'s introduction to cell biology [4] might find no use in reading the corresponding parts herein.

1.1 Computability Theory

Refer to textbooks such as [103] for a detailed treatment of the subjects that are briefly presented here.

Definition 1.1 *An* alphabet *is a non-empty set Σ of symbols.*

Definition 1.2 *A set L of finite strings over a finite alphabet Σ is called a* language *over the alphabet Σ.*

A language may be both finite and infinite. Examples include the infinite language of all strings consisting purely of $\mathtt{a}$s, $L_0 = \{\mathtt{a}^n \mid n \in \mathbb{N}\}$, and the finite language consisting of all strings from the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ of length two: $L_1 = \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$.

Languages can be described by formal grammars:

Definition 1.3 *A* formal grammar *is a 4-tuple $\langle N, \Sigma, R, S\rangle$ where N is a non-empty finite set of non-terminals; Σ is a non-empty finite set of terminals where $\Sigma \cap N = \emptyset$; R is a finite set of production rules of the form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$; and $S \in N$ is the start symbol.*

The formal grammars are ordered in the *Chomsky hierarchy*, denoting the relative complexity of the grammars. Grammars higher in the hierarchy are able to express everything that the lower grammars can.

The following definition will be useful later on:

Definition 1.4 *The number ω is the smallest infinite ordinal number. Thus: $\forall n \in \mathbb{N} : n < \omega$.*

Remark 1.1 Throughout this thesis we will be differing between "hard" mathematical definitions and definitions that are more loose. They shall be called "Definitions" and "Concepts," respectively.

Concept 1.1 *An* automaton *is a machine that is capable of self-operation, i.e., one that, once given an input, requires no intelligent intervention from its exterior in order to process the input.*

1.1.1 Deterministic Finite Automata

Definition 1.5 *A* deterministic finite automaton *is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ where Q is a non-empty finite set of states, Σ is a finite alphabet, $\delta\colon\ Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is a set of accepting states.*

A deterministic finite automaton (DFA) is an abstract formalization of a machine with finite memory. Due to the finiteness of their memory, they are only able to recognize the most restricted class of languages: the regular languages. A regular language can be described by a regular grammar.

Definition 1.6 *A* left regular grammar *is a formal grammar where the production rules are of one of the following forms:* $\{N \to \Sigma, N \to N\Sigma, N \to \epsilon\}$ *where* ϵ *is the empty string. A* right regular grammar *is defined analogously, with* $N \to \Sigma N$ *instead of* $N \to N\Sigma$*. A* regular grammar *is one that is either left or right regular.*

The regular languages are at the bottom of the Chomsky hierarchy, and the DFAs can therefore be regarded as being the simplest (non-trivial) automaton.

1.1.2 The Turing Machine

Definition 1.7 *A* Turing machine *is a 7-tuple* $\langle Q, \Gamma, \Sigma, \delta, q_0, q_{accept}, q_{reject} \rangle$ *where* Q *is a non-empty finite set of states;* Γ *is a finite alphabet with the* blank symbol $b \in \Gamma$*;* $\Sigma \subseteq \Gamma \setminus \{b\}$ *is a finite input alphabet;* $\delta\colon\ Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is a transition function;* $q_0 \in Q$ *is the initial state;* $q_{accept} \in Q$ *is the accepting state; and* $q_{reject} \in Q$ *is the rejecting state,* $q_{reject} \neq q_{accept}$*. The machine has an infinitely large string of symbols from* Γ *as its memory and input/output mechanism.*

At any point in time, exactly one symbol is active, meaning that it is the input to the transition function. Furthermore, the Turing machine is always in exactly one state, which can only be changed by the transition function.

Intuitively, a Turing machine consists of an infinitely long tape with a moveable read/write head on top. The head can read and write a symbol from a cell on the tape, alter the state of the Turing machine based on its internal rules, and move either one cell to the left left or one cell to the right right, where it reiterates the process. When the machine transitions into one of the special states in F, it comes to a complete halt.

A Turing machine has the property that it is possible to make one machine that can simulate any other Turing machine. It is said to compute the function $\tau\colon\ \Sigma^* \to \Gamma^*$ if it halts with exactly $\tau(x) = y \in \Gamma^*$ on its tape on every input $x \in \Sigma^*$. As the tape is infinitely long, the blank symbol is used to represent "nothingness" on the infinitely many unused tape cells. The Turing machine always starts with its tape filled with blanks on all cells that have not been given an input.

1.1.3 Automaton Behaviour

Started with an input string, an automaton either *halts* after a finite number of steps, finishing in an accepting or rejecting state, or it continues forever.

Definition 1.8 *An automaton that always halts in an accepting state when given an input string* $s \in L$ *and never halts in an accepting state when given an input* $s' \notin L$ *is said to be a* recognizer *for the language* L*.*

Definition 1.9 *An automaton that always halts in an accepting state when given a string* $s \in L$ *and always halts in a non-accepting state when given a*

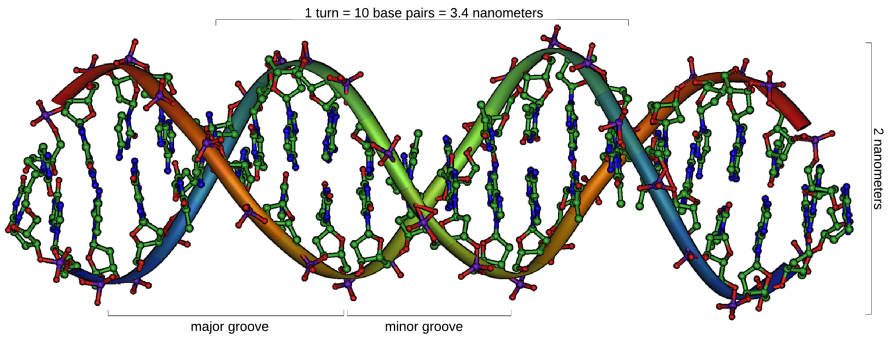


Figure 1.1: *Two single strands of DNA form a double helix because of the hydrogen bonds between the base pairs. The backbones (solid lines) contain stronger covalent bonds, so the bonds between the two strands break easier than the strands themselves. Image from Wikimedia Commons [108].*

string $s' \notin L$ is said to be a decider *for the language L.*

As can be seen from the two definitions above, the difference between a decider and a recognizer is that the decider can guarantee that it always halts.

A language that is recognized by a Turing machine is said to be *recursively enumerable*. Moreover, if the language is decided by the Turing machine, i.e., if the Turing machine is guaranteed to halt eventually, the language is *recursive*. The class of the recursively enumerable languages is at the top of the Chomsky hierarchy. Consequently, if we have a Turing machine, all other automata can be simulated on that.

A DFA is a decider for a regular language; it consumes one symbol at a time until no more remain on the input, leaving the automaton in an accepting or a rejecting state.

1.2 Necessary Molecular Biology

A disclaimer: The author is a computer scientist, not a biologist. The following is supposed to serve the purpose of introducing other computer scientists to the biological concepts used, and as such it runs the risk of being construed too simple by a biologist. Terms and concepts are introduced in a less formal way than that used in conjunction with theoretical computer science, as different traditions and methods exist in the two disciplines, reflected in the language used about and in them.

For a more in-depth introduction, the book by Alberts et al. [4] is recommended.

1.2.1 Deoxyribonucleic acid — DNA

Deoxyribonucleic acid (DNA) is used in the cell to store genetic information, lending itself to be regarded as "storage" in a computer scientist's terminology.

Under normal circumstances, DNA has a helical structure, referred to as the "double helix," see Figure 1.1. Such a molecule is also called double stranded

DNA, as it consists of two single strands intertwined. The two strands of the helix each consists of a sequence of nucleotides, which are sugar-phosphate molecules with a base attached to each of them. There are four different bases, each having a one-letter acronym: Adenine (A), cytosine (C), guanine (G), and thymine (T). Because the sugar-phosphate molecules in each nucleotide are always the same, a DNA strand is described entirely by a string from this four-letter alphabet. The nucleotides attach to each other through covalent bonds* between the sugar-phosphate components. These occur between the fifth and the third corner of the pentagonal sugar component, giving directionality to a DNA molecule: The end with the exposed fifth end is denoted 5', and the other end is denoted 3'. When writing the letter-sequence of a DNA strand, it is always done in the 5'-3' direction (unless otherwise noted). Two strands, x and y, of DNA are *Watson–Crick* complementary if the 5'-3' direction of strand x matches the 3'-5' direction of strand y, only with T interchanged with A, and C interchanged with G. The following demonstrates a short segment of double stranded DNA, written with the four-letter alphabet and directionality:

5'-ACGTAACGGTC-3'
3'-TGCATTGCCAG-5'

The base pairs (A, T) and (C, G) form hydrogen bonds† with each other, thereby causing two Watson–Crick complementary DNA strands to be bonded and form the double helix. The helical structure stems from a slight twist between each nucleotide. A short segment of single stranded DNA is called an *oligonucleotide*.

When heating double stranded DNA, the two strands separate. Due to the difference in the strength of the hydrogen bonds and covalent bonds, each strand remains intact, it is only the bonds between the two that are broken. This process of "melting" the DNA is called *denaturation*, and the reverse process, slowly cooling the single stranded, Watson–Crick complementary DNA to form a double helix again, is called *hybridization*.

1.2.2 Ribonucleic acid — RNA

Closely related to the DNA molecule is the *ribonucleic acid* (RNA). It consists of three of the same four bases, where the fourth, thymine, has been replaced with uracil (U). In the cell, RNA plays a role in the intermediary steps between the DNA and the proteins that the DNA code for.

RNA can be *transcribed* from a DNA strand. When this happens, only a single strand is synthesized. Hence, RNA is usually single stranded, and therefore not forming the double helix. Instead, RNA can fold itself into various three-dimensional shapes known as the *tertiary structure*, enabling it to have more versatile uses than the DNA molecule. This, combined with a less stable backbone structure, causes RNA to be less stable than DNA.

RNA is categorized into different types depending on its use, including transfer RNA (tRNA), used to supply amino acids to the process that builds

*A covalent bond is formed when two atoms share one or more electrons. The number of shared electrons influence the three-dimensional structure of the compound.

†A hydrogen bond is formed when a hydrogen atom is "shared" between two atoms, i.e., it is exerting electrical influence over both atoms. Hydrogen bonds are weaker than covalent bonds.

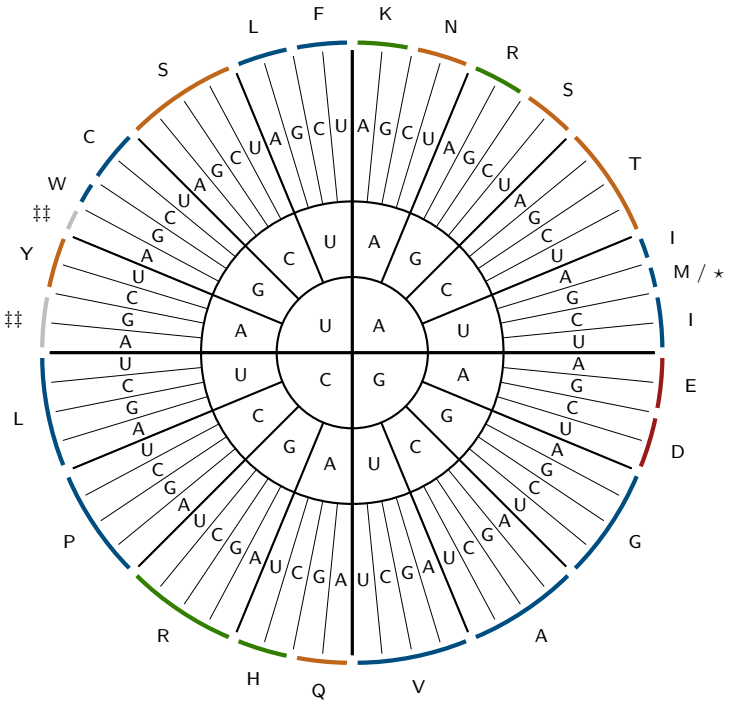


Figure 1.2: *Written from the center to the periphery are the codons of RNA, along with the one-letter names of the 20 amino acids that they each code for. The codons for ‡‡ are stop codons that signal the end of a sequence of amino acids, and the codon AUG functions both as an initiation symbol that starts a protein-coding sequence, and as the codon for methionine (M). The coloring indicates the type of amino acid: negatively charged are red, positively charged are green, uncharged are orange, and nonpolar are blue. Based on an example from [60].*

the proteins (see next section); messenger RNA (mRNA), transcribed from the DNA and used as the code for a protein; and ribosomal RNA (rRNA), used as a catalytic component in the process known as *translation* from mRNA to amino acid chains. The translation of RNA to amino acids happens in triplets: Three RNA bases code for an amino acid, illustrated on Figure 1.2. Due to its combined storage and catalytic capabilities, RNA has the interesting property that it is in theory able to self-replicate; a property that neither DNA nor proteins share.

1.2.3 Proteins

A *protein* is a sequence of amino acids attached to a backbone structure, similar to the structure of DNA. However, where DNA almost always folds into the same helical shape, the shape of different proteins varies, being determined by the properties and sequence of the amino acids which they consist of.

20 different amino acids are used as the building blocks for proteins, although several more exists. They are grouped in two main categories: the polar and

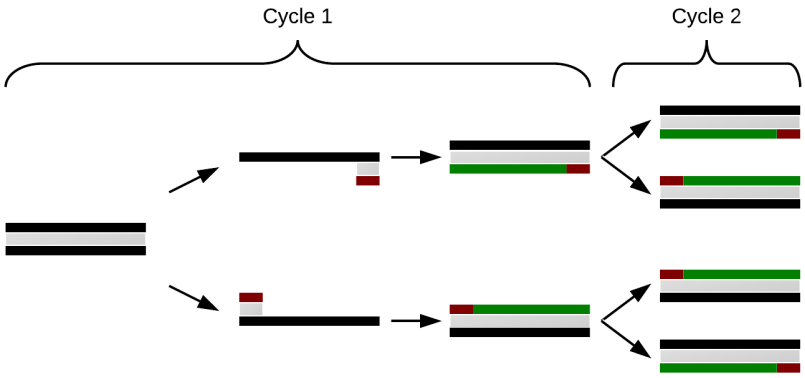


Figure 1.3: *Two cycles of PCR. After each cycle, the amount of DNA is doubled. The short red lines indicate the DNA primers, used by the polymerase enzyme to locate where the DNA helix should be restored. In this stylized example, the entire DNA strand is amplified, but it can also be done with a specific subpart of the original DNA strand.*

the nonpolar amino acids. The polar amino acids are further categorized, being grouped as negatively charged, positively charged, or uncharged.

The groups of amino acids influence the shape of the protein molecule in different ways; the negatively and positively charged polar amino acids cause electrostatic attractions between each other, the uncharged polar amino acids can form weak hydrogen bonds, and the nonpolar are more rigid and do not form bindings to other amino acids. A third force called the *van der Waals attraction*[‡] also assists in the formation of the protein. Moreover, in an aqueous environment, the nonpolar molecules tend to be "packed" together because they are hydrophobic and therefore repelled by the water molecules surrounding them — analogous to the behavior of oil in water.

A protein's shape determines its chemical and physical capabilities; some proteins work as a kind of infrastructure, providing the mechanical support for tissue, whereas others serve as catalysts for different chemical reactions. The latter type is called an *enzyme*.

One enzyme that plays an important role in many DNA-based computing models is *DNA ligase*. It is able to "glue," or *ligate*, together two DNA strands. In the cell, among other places, it is used in the repairing of broken DNA strands. Usually, for two DNA strands to be attached to each other by a ligase enzyme, they must have have matching *sticky ends*, which are short pieces of single stranded DNA at the ends of a double stranded DNA molecule.

1.2.4 Polymerase Chain Reaction

Due to the Watson–Crick complementarity, each strand of a DNA double helix contains the same information, albeit in different representations. Consequently, with only one of them it is possible to restore the original helix. This property

[‡]A bonding between atoms when they are very close to each other, stemming from fluctuating electrical charges between the atoms.

is used in the process known as *polymerase chain reaction* (PCR), which is used to amplify DNA, i.e., increase the amount of certain DNA strands in a solution.

Each cycle of the process works by separating DNA into single strands by heating them, letting a short piece of DNA (a *primer*) attach to each of the two DNA strands, and letting a *DNA polymerase* restore each single stranded DNA to a double helix. A DNA polymerase is an enzyme that, when given a single stranded DNA with a short double stranded section, rebuilds the DNA and reestablishes its double-helical structure, beginning from the double stranded section. The primer forms the short double stranded section, and thus serves the purpose of guiding the polymerase.

A cycle can be repeated as needed, enabling the production of 2^n copies of the original DNA upon n cycles. As the output DNA of cycle n is treated in cycle $n+1$, the process is called a chain reaction. Figure 1.3 is a stylized representation of the first two cycles of a PCR process.

1.2.5 Restriction Enzymes

The special class of enzymes called the *restriction enzymes* can cut single or double stranded DNA at specific short segments of bases in the DNA strand, known as the recognition sites. A "cut" is a breaking of one of the covalent bonds between the sugar-phosphate backbone links on each side of the double helix, and possibly the breaking of hydrogen bonds between base pairs. The latter happens only if the restriction enzymes leave a sticky end.

For example, the restriction enzyme *FokI*, the uses of which are discussed later, recognizes the sequence GGATG and its complementary strand, and cuts the DNA strand 9 and 13 base pairs to the right of the recognition site, thereby leaving a sticky end [58]. Using X as a placeholder for any of the four bases, it looks like the following, where the molecule is split at the spacer between the black and the red part:

5'-XXXXX**GGATG**XXXXXXXXX XXXXXXXX-3'
3'-XXXXX**CCTAC**XXXXXXXXXXXXX XXXX-5'

1.2.6 Gel Electrophoresis

Gel electrophoresis is a technique used to sort DNA strands based on their lengths. The strands are placed in small wells on an agarose gel,[§] and an electric field is applied. Because the DNA molecules are negatively charged, they move through the gel, which acts as a hindrance or "sieve" for the molecules, letting smaller molecules pass more easily. The distribution of lengths of the DNA strands in a solution can therefore be approximated by inspecting how far in the gel the strands have migrated. This is done with the help of a special "marker well," containing a sample of DNA strands with known sizes.

After performing an electrophoresis, a part of the gel can be cut out, and the DNA present in it can be used again. It is therefore a way to extract DNA strands with specific lengths out of a solution.

[§]Agarose gels are made from polysaccharides obtained from red algae.

2 Literature Survey

The intent of this chapter is to explore and describe a relevant and representative subset of the literature on the subject of biomolecular computation. The research has been categorized into the two categories "dry" and "wet" methods.

The subject is broad and the articles many. More thorough surveys are presented in [13, 36, 46, 50, 52, 63].

2.1 Dry Methods

We use the classification "dry" to denote those methods that are inspired by natural, biological, or biomolecular processes, but not implemented in a biomolecular substrate.

2.1.1 Cellular Automata

One method of computation having Nature as its inspiration is the *cellular automaton*. It was introduced in the 1940s by John von Neumann [116] and has been shown to have universal computing capabilities [20].

A cellular automaton consists of an n-dimensional grid with infinitely many cells and a finite set of states for each cell. Cells change their state according to a finite set of rules; for example, a cell can switch from "on" to "off" if it has got no neighbors, anthropomorphically interpreted as "loneliness."

2.1.2 Neural Networks

The man-made computing devices have been compared with the capabilities of the human brain since at least the late fifties [115], and mathematical models for logic based on the human nervous system were proposed as early as 1943 [75]. In the hope of 1) identifying the exact processes taking place in the brain, and 2) using those processes to build better computers, the field of *neural computation* evolved. The two-sided structure of the ambitions of this field has caused it to split into two; brain theory deals with the first aspect, while the theory of *artificial neural networks* deals with the latter.

An artificial neural network is a mathematical abstraction that seeks to model the way "real" neurons in the brain work. A network is thus a set of interconnected neurons, each with a set of inputs and one output. The output of one neuron can be the input of another, hence the network structure. Mathematically, a neuron is a function of n inputs with n associated weights $w_1 \dots w_n$. The behavior of the network is highly dependent on the associated weights, and an important part of the power of neural networks is their ability to "learn" these weights during a training phase. A neural network can therefore be described by its network topology, its primitive functions that are performed by the neurons, and the method used for adjusting the weights in order to attain optimal performance.

2.1.3 Evolutionary Computation

Inspired by Darwinian evolution [34], the field of *evolutionary computation* arose [8]. This field seeks to obtain efficient and correct computation through a more "high-level" simulation of Nature: Whereas the idea behind neural networks and cellular automata is to simulate small, separate entities, evolutionary computation performs the simulation on larger systems of interconnected entities, each one not necessarily being modelled after Nature. It can be divided into four subfields [36]: *genetic algorithms* [59], *evolution strategies* [21], *evolutionary programming* [42], and *genetic programming* [69].

The field is called "evolutionary" because the different methods that can be categorized under this umbrella all share the same basic algorithm: A population of possible solutions is created; a series of mutations, crossings, and mixings between solutions is performed; and natural selection, represented by a *fitness function*, is used to pick out the best ones of a generation that will multiply and survive. This is repeated several times, causing the later generations to consist of more "good" solutions than the earlier. The crux of the method is the use of *variation* and *natural selection* to perform *adaptation* of the solutions to the environment. Hence, the difficulty of using the method lies in choosing a correct measure for fitness and a correct fitness function.

2.1.4 Swarm Intelligence

Another approach whose power comes from high-level synthesis of the behavior of several seemingly simple components is the discipline *swarm intelligence*. The term was coined in the field of robotics in the late 1980s [18], and the motivation behind it is the emergence of complex behaviors in societies of simple individuals, e.g., ants, birds, or fish.

Studies of the ant type *Iridomyrmex humilis* revealed an ability to quickly find the shortest route to a food source by laying pheromone trails [48]. Inspired by this, a model with artificial ants laying artificial pheromone trails has been devised to generate solutions to the traveling salesman problem: The ants could iteratively find shorter paths through the graph using the fact that the pheromone concentration is lower on longer trails [38]. Repeating the algorithm caused the solution estimate to become better, i.e., shorter.

Similarly, the flocking behavior of birds has been studied, and an emulation has been devised in which each "bird" shares three simple behaviors: Avoid collisions, maintain the same speed as the others, and stay close to them. The complex flocking behavior emerged in the interplay of several birds following these rules [93]. Recently, Chazelle has studied the convergence rates of flocking networks, viz., the time it takes for the bird flock to agree on a direction and speed. He found both a high upper and lower bound, as a "tower-of-twos" function of the number of steps in the simulation, defined by:

$$2 \uparrow\uparrow 1 = 2$$
$$2 \uparrow\uparrow n = 2^{2 \uparrow\uparrow (n-1)} \text{ for } n > 1.$$

Chazelle showed that the number of steps required for a flock of n birds to reach a steady state is upper-bounded by $2 \uparrow\uparrow O(n)$ and lower-bounded by $2 \uparrow\uparrow \Omega(\log n)$ [29]. Given these bounds, the method must be said to be impractical, and, if used, it should employ some kind of approximation technique.

2.1.5 Theoretical Models for Biomolecular Computation

Geffert showed a normal form theorem in [47] stating that any recursively enumerable language can be generated by a phrase-structure grammar consisting of five nonterminals, three context-free rules, and two context-sensitive rules. This property was used by Yokomori [133] to construct a computation model based on DNA self-assembly (denoted "grammar-based computation of self-assembly" in [52], and "YAC" in [133]). The model works by constructing DNA molecules in a clever way, such that Geffert's grammar rules and the way the DNA strands hybridize behave equivalently. Yokomori showed that any recursively enumerable language can be recognized by this model.

Another approach is based on *equality machines* (EM) [39]. An EM is an automaton with one read-only input tape; two write-only, one-way output tapes; and a set of states. The machine behaves according to a transition function, as a Turing machine, but it accepts if and only if the contents of the two output tapes are the same in the terminating state. Engelfriet and Rozenberg showed that the class of languages recognizable by a non-deterministic EM is the class of recursively enumerable languages [39]. Yokomori and Kobayashi gave a model for DNA computation and showed that it can simulate any EM [134].

Kobayashi suggested to use a subset of Horn clause logic [61] called simple Horn programs as a basis for a DNA implementation of a computer [67]. A simple Horn program consists of a finite set of Horn formulas in the format $\forall X_1, \ldots, X_m \colon (F \leftarrow F_1 \wedge \ldots \wedge F_n)$, where each F and F_i are Horn clauses: $A(X_1, \ldots, X_k)$. They are called ground atoms when $X_1 \ldots X_k$ are constants. Informally, the computation in simple Horn programs is performed by iteratively reducing a rule's body to its head, provided that the body evaluates to true. This starts from a set of statements known to be true, i.e., axioms. Kobayashi gave a method to implement this reduction function, called the *immediate consequence operator*, using whiplash polymerase chain reaction (whiplash PCR, see Section 2.2.1.4) to perform the equality checking and parameter copying. It is necessary to construct two types of specialized DNA strands; one for the constant symbols and one for each of the predicate symbols. Further investigations of Horn clause logic and DNA molecules are explored in [113].

Boneh with colleagues gave a small taxonomy of available DNA strand operations when designing algorithms, such as "extract" and "amplify" [22]. They presented some theoretical results concerning the problems *Circuit-SAT*, *MAX-Circuit-SAT*, and *Regular-Circuit-SAT*, proving that they are all solvable using a DNA-based computer with the operations from their taxonomy.

Regev et al. suggested the use of the process calculus known as π-calculus [76] to model biological systems [92, 91, 90]. This was inspired by Fontana, who in 1996 discussed how "to develop a formal understanding of self-maintaining organizations" [43]. Other attempts to formally model biological systems have been Danos and colleagues' κ-calculus [33], which attempts to model protein behavior, and Cardelli's "DNA strand algebra" [27].

Inspired by cell membranes, Păun studied a model of computation known as *membrane computing* in 1999 using a formalism that he called the *P-system* [84, 85]. Ardelean et al. studied the feasibility of realizing the model using bacteria [6, 7].

Head introduced the *splicing system*, alternatively known as the *H-system*, in 1987, thereby describing a formal model for the manipulation of DNA strands,

for instance by cutting them with enzymes [56]. This model was shown to be able to simulate any Turing machine [32].

2.1.6 String Encoding

In several of the articles referenced in this chapter, the authors speak about encoding information, such as solution candidates or machine states, on DNA molecules. The actual coding schemes used are not described, but it is implied that "some" practical scheme is used, which minimizes the amount of unwanted and unstable hybridizations in the DNA. Boneh et al. discussed one way to represent binary strings using DNA strands with 30 bases (30-mers) [22]. The idea is to assign each bit value on each position a unique 30-mer strand and joining them together with a special separator strand. To minimize the amount of unwanted hybridizations, one must minimize the length of the longest common substrings between any two parts. The authors suggest using "some good code," or using a randomized method [22].

2.2 Wet Methods

Contrary to the "dry" methods, the "wet" methods are the ones that have been constructed in a laboratory, using some biomolecular technique.

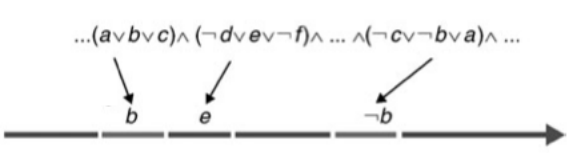
2.2.1 DNA-Based Computing

The research area *DNA-based computing* concerns interactions between DNA strands as the basis for computation. A biomolecular toolbox of considerable size exists for the manipulation and amplification of DNA molecules, a fact that could explain the apparent preference over other models that DNA-based computing enjoys. An early motivation for this approach was Bennett's observation of the possible energy efficiency of that type of computation [19].

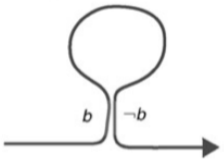
2.2.1.1 Solutions to NP-Complete Problems

The first breakthrough in computing with DNA strands was in 1994 with the publication of Adleman's paper [3], demonstrating a method to compute a solution to an instance of the NP-complete *directed Hamiltonian path problem*. Adleman noted that the computation rate of the DNA strands in his test tubes were orders of magnitude higher than those of the fastest supercomputers of the time, at least if each ligation reaction was viewed as an instruction. The technique he used simulated a non-deterministic Turing machine, in the sense that he started out with generating every possible solution, encoded on single stranded DNA, whereafter the incorrect ones were removed, and it was checked whether any correct solution candidates remained. This was accomplished through several "washes," performed in the laboratory over a period of seven days. Lipton showed how to extend the method demonstrated by Adleman to NP-complete problems in general [70].

Since then, several other NP-complete problems have been solved using techniques similar to those of Adleman's. In 1997 Ouyang and colleagues solved an instance of the *maximal clique problem* by encoding each possible clique in a particular graph as binary digits on DNA strands, removing the incorrect ones



(a) *Solution candidate on a single stranded DNA molecule.*



(b) *The impossibility of the solution becomes evident when a hairpin is formed.*

Figure 2.1: *Illustration of a hairpin formation on a single stranded DNA molecule. Images from [52].*

that contain vertices that are not connected, and picking out the largest [83]. As in Adleman's approach, this technique utilizes the high degree of parallelism inherent in a "soup" of DNA molecules in a test tube, thus allowing for simulation of a non-deterministic Turing machine.

In 2002 Braich and colleagues solved a 20-variable instance of the 3-SAT problem, finding the unique solution amongst the 2^{20} possibilities [23]. Their algorithm worked by repeatedly refining solution candidates by "sorting" them using gel electrophoresis performed for each clause. They constructed an electrophoresis box in such a way that strands satisfying the clause in question were captured one place, while the rest ended in a reservoir in another place. The problem instance they solved was larger than the one solved in the work of Sakamoto et al. from 2000, where they computed the solution to a six-clause, ten-variable instance of 3-SAT using a technique employing *hairpin formation* [98]. Solving the problem with hairpins, one generates every possible truth value assignment, where at least one variable per clause is assigned "truth." The negation of the variable a, $\neg a$, is encoded as the Watson–Crick complement of the encoding of a. Thus, if a strand contains both a and $\neg a$, those two parts will hybridize, forming a "loop," or hairpin, on the single stranded DNA molecule (see Figure 2.1). Hence, eliminating every hairpin formation or greatly increasing the percentage of non-hairpins in a pool will allow the correct solution to be observable with gel electrophoresis. The authors note that their method has the benefit of only requiring a constant number of laboratory steps, regardless of the size of the problem. However, they also note that they require much more DNA to perform the computation: 3^m strands are generated for m clauses, as opposed to Adleman's 2^n strands for n variables [98].

Liu with colleagues demonstrated a solution to the *graph coloring problem* in 2002 [72]. The approach taken in that paper follows the same high-level algorithm as the maximal clique solution in [83], insofar that all possible colorings were generated, the incorrect ones removed and the minimum singled out. The authors remark that the method is limited with respect to the problem size; the exponential increase in the pool size renders it impractical to process problem instances larger than 30 vertices.

An algorithm that solves the *subset-sum problem* using DNA computing methods was presented by Chang and colleagues in 2004 [28]. For this they developed an n-bit parallel adder and an n-bit parallel comparator, demonstrat-

ing a stronger version of DNA addition than first showed in 1996 by Guarneri et al. [49].

In [120] an algorithm for solving integer linear programming problems using DNA hybridization is described, but the authors do not apply the algorithm in an experiment.

2.2.1.2 Data Storage

In [3] it is remarked that the data storage capabilities of DNA molecules are enormous — 1 bit per nm^3. In comparison, Adleman notes, a videotape can store 1 bit per 10^{12} nm^3. Shoshani et al. mention that the storage capacities of 1 gram of DNA in 1 cm^3 is approximately 750 terabytes [101]. Cox discusses ways to use DNA as a long-term storage medium in [31], and a memory model based on DNA structures was presented by Kashiwamura et al. in 2005 [65].

Recently, the field of *infochemistry* has emerged, focusing not on the computational aspects of chemistry, but on the possibilities for storage and transmission of information [112].

2.2.1.3 Complexity

A problem noted in Adleman's paper [3] is, that the mechanics of the DNA "computer" work in a less predictable way than that of the classic machines. Hybridization can result in false positives or negatives due to reactions between non-matching bases followed by PCR amplification [37]. Work has been done to investigate how to compute reliably despite such noise [86, 64].

2.2.1.4 Methods

A number of different methods for DNA computing have been proposed.

Adleman–Lipton method. The earliest type of method demonstrated was by Adleman in 1994 [3] and reformulated for more general use by Lipton in 1995 [70], which encodes all possible solutions and removes the incorrect ones. This is done by repeatedly ligating the solution instances to the strand-representation of each vertex, followed by a process that removes those strands which were not ligated to a vertex-strand. In Adleman's original paper he states that the total amount of work required was seven days in the laboratory. The number of laboratory procedures required should grow linearly with the problem size [3].

Boolean circuits. Ogihara et al. have studied the construction of Boolean circuits using DNA molecules [81, 82]. Amos and Dunne demonstrated a method for simulating a circuit more efficiently than Ogihara's initial suggestion [5]. Later, Stojanovic and Stefanovic created the logic gates AND, NOT, and XOR using enzymatic reactions on DNA [106]. They used this technique to build a circuit that plays tic-tac-toe on a three-times-three board against a human opponent [107]. Using fluorescent molecules, the circuit was able to mark its moves in response wells, corresponding to the fields of the board. In 2004, Su and Smith constructed a NOR gate using DNA molecules [109], and in 2006 Seelig et al. introduced another method to implement logic gates, based on *branch*

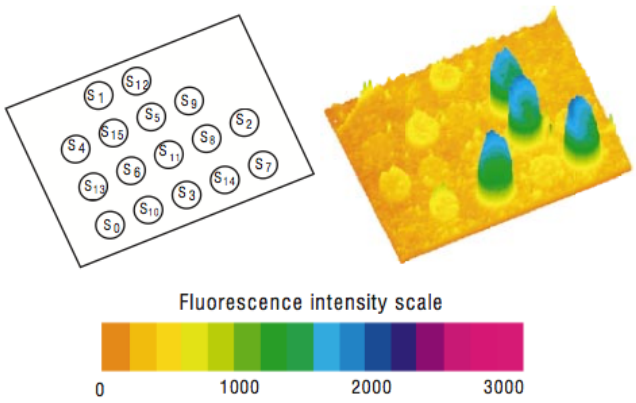


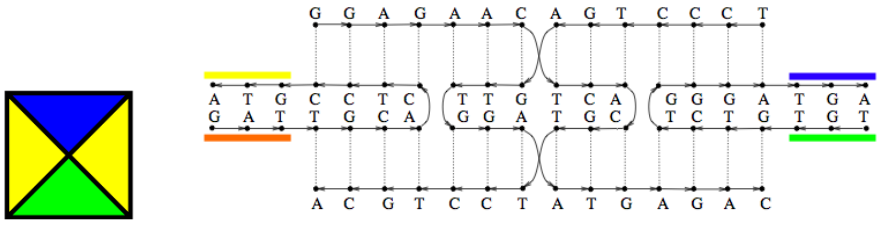
Figure 2.2: *Illustration from [71] illustrating the increased fluorescence at the spots on the gold surface containing the correct solutions.*

migration, i.e., the replacement of specific DNA strands [100]. Qian and Winfree developed the *seesaw gate* based on branch migration, and demonstrated a circuit capable of computing the square root of four-bit numbers [88, 87].

Sticker model. Roweis and his colleagues presented the *sticker model* of DNA computing in 1998 [96]. The model has a random access memory wherein bits are encoded as either single or double stranded DNA. Like other models it utilizes DNA strand separation as the central computing mechanism. Four basic operations on bit strings were described in the paper: combining, separating, setting on, and clearing (setting off).

Whiplash PCR. A technique called *whiplash PCR* as the basis for execution was first described by Hagiya et al. and Sakamoto et al. [51, 99]. Nishikawa and Hagiya implemented a simulator for the technique in [79]. Whiplash PCR is a combination of a "whiplash" intramolecular reaction, in which one end of a single strand folds back onto itself, thereby creating a hairpin structure, and polymerase extension of the double stranded part, thus allowing for state transitions. The state of the machine is represented in the end of the DNA strand — the tip of the whip — and it is therefore a DFA. Komiya and his colleagues performed a wet experiment with whiplash PCR, successfully executing eight successive state transitions [68].

Surface-based computations. Liu et al. solved a simple four-clause instance of the 3-SAT problem with four variables using a technique called *surface-based DNA computing* [71]. The technique was first described by Smith, Frutos, and colleagues in 1997 [104, 45], and it works by fixating possible solutions encoded as DNA strands to a gold surface, applying a mark-destroy-unmark algorithm that removes the incorrect ones, and extracting the answer by inspection of the surface. The "mark" phase hybridizes all but the single strand representations of the incorrect solutions — those allowing $a \wedge \neg a$. The "destroy" phase is executed by an enzyme (*E. coli* exonuclease I) that destroys all unhybridized strands, thereby removing the incorrect solution candidates. Finally, the "unmark"



(a) *Wang tile (from [125]).*

(b) *DNA tile. The colored bands indicate the "sticky ends" (from [130], slightly altered).*

Figure 2.3: *Two different types of representations of Wang tiles.*

operation regenerates the molecules, returning the remaining strands into single stranded form. This was repeated for every variable in the clause. The benefit of this method is the increased ease of readout compared to the other, "test tube based" approaches, because the researchers are able to attach a fluorescent molecule to the remaining correct solutions and bind them to another gold surface, containing a copy of each of the original solution candidate molecules. Thereby, the spots with increased fluorescence indicate the correct solutions, as in Figure 2.2.

Self-assembly. In 1961 the mathematician Hao Wang proposed a system later dubbed "Wang tiles" [119], which informally consists of a finite set of colored tiles. It was later shown to be able to emulate any given Turing machine [95]. The logical equivalence of certain DNA structures with Wang tiles was demonstrated by Winfree et al. in [129, 127, 130, 128], based on the construction of DNA "tiles" consisting of special strand formations having four sticky ends; see Figure 2.3. The configuration of these four ends is the equivalent of the configuration of the colors of Wang tiles. Thus, the computation proceeds by self-assembly of the tiles, provided that they are designed properly [25]. Patterns for DNA tiles have been designed to emulate an XOR gate [73], and Feldkamp et al. present a compiler that translates formal grammars into DNA capable of structured self-assembly [41].

More recently, Yin et al. have developed a model and pictorial language called *reaction graphs*, depicting a high-level view of the way DNA molecules can interact [131]. In their paper, they describe the path from a desired function through its formulation as a reduction graph, to the construction of "secondary structures" — logical representations of DNA molecules — and finally the actual DNA strands. As the nodes of a reaction graph are abstract representations of hairpin structures, an "execution" of a reaction graph corresponds to a specific sequence of unfoldings or foldings of DNA hairpin structures. They give an example of a DNA molecule with a structure resembling a binary tree, constructed using a reaction graph with ten nodes.

2.2.2 RNA-Based Computing

Regulatory behaviors performed by processes in live cells with different types of RNA as an integral part are described by Isaacs et al. and Davidson et al.

in [62, 35]. These processes can be interpreted as the evaluation of Boolean logic formulas [110]. Furthermore, the similarity between RNA and DNA molecules have led researchers to believe that RNA has a similar potential as a basis for computing [57].

Benenson presents a brief survey of some of the newer RNA-based approaches to biomolecular computing in [13, 14]. A major reason for some researchers' preference of RNA over DNA in molecular computing is that RNA can readily be synthesized in live cells, arguably improving the chances of building actual "computers" for medical purposes in live cells [13, Section 4.3]. The methods discussed in [14] focus on the construction of logical circuits in either conjunctive or disjunctive normal form. Heitsch and colleagues describe methods to design RNA strands with desired spatial structures in [57]. A more general discussion of strand design is given in [74].

An early result using RNA as the basis for computation was by Faulhammer et al. [40], who solved an instance of the *Knight problem*: A special case of the SAT-problem formulated as a chess problem. They used hybridization between RNA and DNA as the basis for computation, removing the incorrect solutions by applying an enzyme (*ribonuclease H*), similar to the technique of Liu and colleagues [71]. Their reason for using RNA and not DNA was scalability, because the enzyme allows for easier solutions to larger problem instances than the DNA-cleaving enzymes offer [40, 97].

In [94, 126] different approaches to the creation of Boolean logic processing using RNA components are described.

2.2.3 Protein-Based Computing

A DFA has been built using DNA molecules and the restriction enzyme *FokI* by Benenson et al. [17, 15]. In their experiment they managed to construct a DFA which had its states and symbols encoded on short DNA strands (oligonucleotides) and the actual state transitions performed by the restriction enzyme. Every ⟨state, symbol⟩-pair was encoded as oligonucleotides, this being the "software" of the machine, and the input string was encoded on double stranded DNA with six base pairs for each symbol. This strand also contained the state of the machine. To control the restriction enzyme, the transition function of the DFA was encoded as a set of "transition molecules," implemented by partially hybridized DNA strands. The computation proceeded by repeatedly cleaving the input strand at a special marker site, causing the transition molecule representing the current ⟨state, symbol⟩ to hybridize with the cleaved input strand, thus resulting in a new recognition site being exposed and a new enzymatic reaction; see Figure 2.4. In their article, the authors report that 10^{12} DNA automata ran independently in parallel, all encoding the same DFA, with a combined transition rate of 10^9 per second. The start of the computation was when the different components were mixed together. After this step, no human interaction was needed until the result of the computation was ready to be read out with gel electrophoresis. The inherent parallelism of the machine was demonstrated when the authors tried to add two different inputs simultaneously and the machine produced two different outputs.

The group presented a medical application of their molecular DFA in 2004 [16]. The idea is to use a DFA like the one described in [17], but constructed in such a way that it releases a drug when it enters its accepting state,

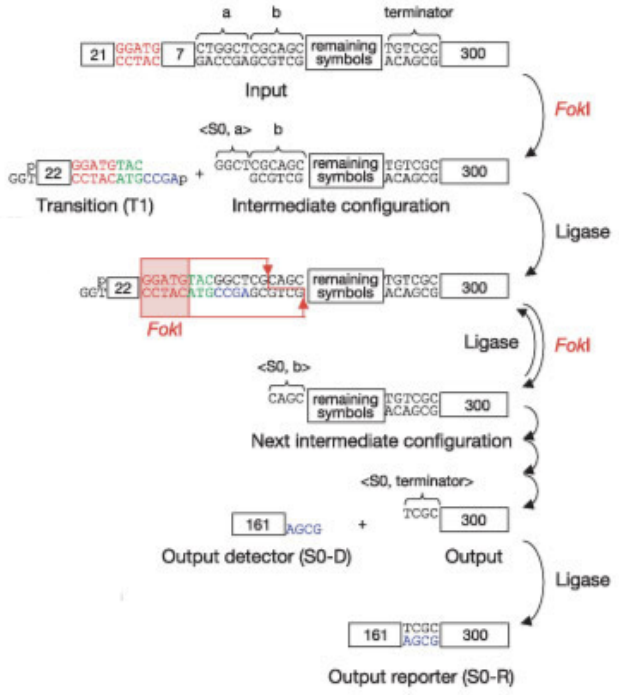


Figure 2.4: *Illustration of a DNA-based DFA in the process of computing; the state/symbol strands are chopped off consecutively by* FokI*, ultimately leaving only the special accepting terminator strand (from [17]).*

conditioned on the presence or absence of certain genes. This is accomplished by constructing the automaton as a hairpin structure, where the hairpin-loop encloses a drug and the stem represents a logical conjunction of clauses that must be fulfilled in order for the drug to be released. The clause checking proceeds as the DFA, with an additional mechanism for checking for the presence of certain genes. The latter is performed by having a "team" of DNA molecules for each gene type that must be checked, constructed in such a way that they hybridize and form a recognition site for *FokI* in the presence of a certain gene. Thus, for each fulfilled condition a part of the hairpin-stem is cleaved by *FokI*, with the result that the hairpin opens and the drug is released only if every condition holds.

Recently, Ran et al. implemented a system capable of answering simple logic queries (including the ubiquitous question regarding Socrates' mortality) that utilizes the *FokI* enzyme [89].

Shoshani et al. demonstrated another variant of the *FokI*-technique in 2010, implementing a 2-state, 2-symbol DFA [102]. The team essentially attached genes coding for green and cyan fluorescent proteins to the special marker molecules that each attach to the ⟨state, symbol⟩-strands representing final states S_0 and S_1, respectively. The input strand and detector strands were

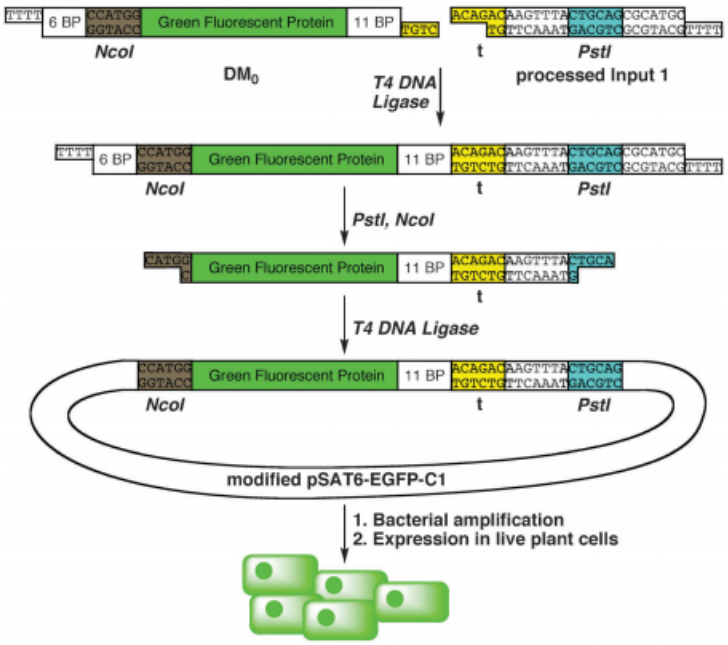
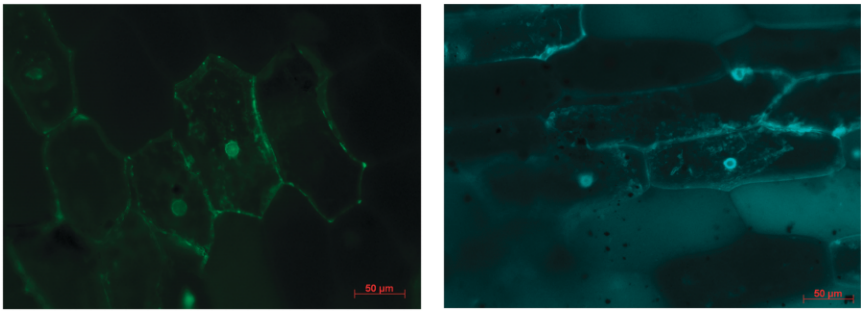


Figure 2.5: *The final step in the DFA computation from [102]. This part comes right after a sequence of computation steps, constructed like the one illustrated in Figure 2.4. The final output detection molecule* DM_0 *containing the gene coding for the fluorescent protein is restricted by* NcoI *and* PstI*, thereby making both ends of the strand sticky, enabling it to be ligated with another strand to form a circular plasmid that can be inserted into cells.*

engineered such that recognition sites for the restriction enzymes *NcoI* and *PstI* were present on the final, detected state. This allowed the "solution strands" to be cleaved with *NcoI* and *PstI*, thereby making them ligate with a special DNA strand which caused them to become circular plasmids, see Figure 2.5. This plasmid was then amplified in colonies of *E. coli*, whereafter it was inserted into cells from the onion plant. The expression of the fluorescence protein genes in the plasmids resulted in green fluorescent onion cells for S_0 and cyan fluorescence for S_1, see Figure 2.6. The authors note that this demonstrates the method's ability to alter *in vivo* cells.

The construction of Boolean logic gates using proteins has been studied by different groups. Willner and colleagues used specific chemical compounds, among others glucose, to construct a network of enzyme-based logic gates, each gate itself composed of one or more enzymes [11, 78]. Using these, they constructed a half-subtractor and a half-adder [11]. In [114] the authors discuss a method to implement a NAND gate using specifically designed proteins, with DNA strands acting as "wiring" between the gates, essentially pairing each connected output and input port between two gates with two Watson–Crick complementary DNA strands.

Coupling the enzyme *RecA* with single stranded DNA, Bar-Ziv et al. demonstrated a stochastic automaton having as its operational basis random ligation



(a) (b)

Figure 2.6: *The onion cells fluoresce green (a) or cyan (b), depending on the final state of the DFA; the plasmids resulting from each state contain the gene for the appropriate protein. From [102].*

and disassembly (that is, denaturation) [9]. By running several automata (10^5–10^6) in parallel, the authors were able to observe the behavior of the machines with noise removed.

2.2.4 Bactoputing

The field of *bactoputing*, i.e., computing with bacteria, is about moving the DNA computing mechanisms inside a bacterium or into colonies of bacteria. Baumgardner et al. give three main advantages of bactoputing over DNA based computing [12]:

1. the autonomy of bacteria minimizes the need for human intervention;
2. as bacteria can evolve and adapt, they can adapt to specific problems;
3. a colony of bacteria grows exponentially in size, which can be regarded as an addition of processors to the system.

The term cellular computing is also used [111].

Engineering of *synthetic gene networks* is discussed and reviewed by Weiss et al. in [124, 123]. A synthetic gene network is an attempt to control the behavior of cells, so as to obtain, e.g., logic functions, by assembling components (including DNA, RNA, and proteins [122]) into structures and embedding them in cells. The construction is performed both by conventional "top-down" design [105], employing circuit design techniques, but also by *directed evolution*, in which cells are manipulated to mutate their DNA into genes that code for the desired networks [80, 132].

Haynes and colleagues constructed an *E. coli* bacteria-based computation that solved an instance of the *burnt pancake problem*, a sorting problem where the only allowed operation is to reverse the top k elements of a stack [55]. The authors used an enzyme from *Salmonella typhimurium* that recombines DNA strands by reversing specific parts of it, enclosed by special palindromic markers. Using this, they designed the DNA strands such that those representing correct solutions caused the bacteria to be antibiotic resistant.

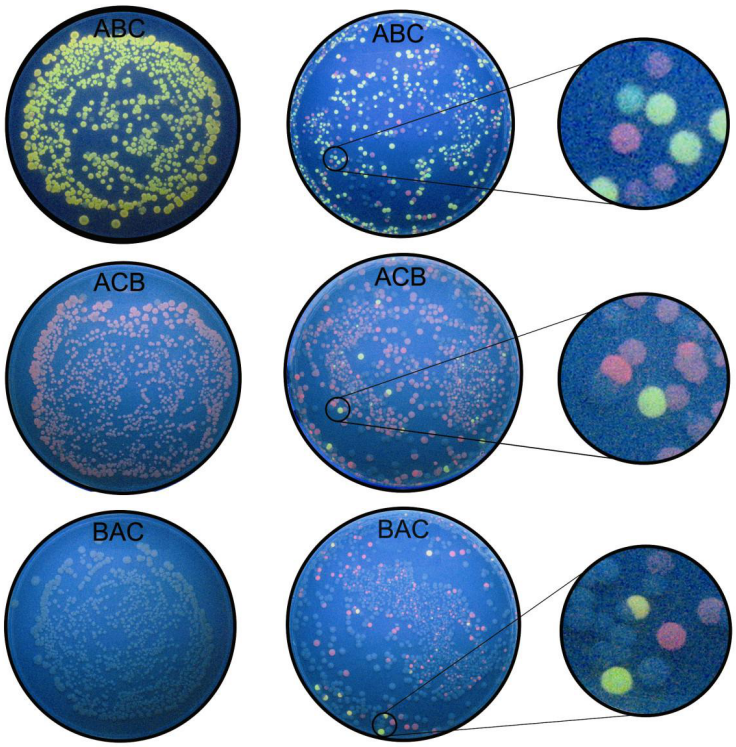


Figure 2.7: *Petri dishes with bacteria colonies encoding solution candidates. The left column contains three different starting colonies, encoding ABC, ACB, and BAC as candidates, respectively. Only ABC is correct, so it fluoresces yellow. The middle column shows the petri dishes after the recombination has been performed, demonstrating that all of them arrived at correct, i.e., yellow, combinations. Image from [12].*

Baumgardner et al. solved a three-node instance of the directed Hamiltonian path problem using colonies of *E. coli* bacteria that were modified with synthetic gene networks [12]. Nodes in the graph were represented using DNA strands in a way similar to Adleman's [3], but were designed using genes that result in specific phenotypes: red and green fluorescence. The same technique as in [55] was employed to ensure that only correct solutions exhibited the desired phenotype, namely the combination of red and green fluorescence, observable as yellow fluorescence (see Figure 2.7). The authors argue that larger instances of the problem can be solved using their method with a linear relationship between the number of genes encoding nodes and the number of nodes.

2.2.5 Other Methods

The *billiard ball model*, a special type of cellular automaton, was introduced by Fredkin and Toffoli in 1982 [44]. This established the field of collision-based computing, which has been studied from the perspective of biological computing. Adamatzky et al. argue that the slime mold *Physarum polycephalum* can be used as implementation substrate of a collision-based computer in [2], following Nakagaki's study of the slime mold's apparent maze-solving capabilities [77].

3 Multiple Kinds of Universality

We will in this chapter work towards a theoretical foundation for quantifying the nature of the particular qualities that make some machines, or computers, capable of doing the work of others. To put it bluntly: what makes the simulation of other computers by means of software possible? The extent to which this capability is a requirement for a "practical" computer, suited for real-world computational tasks, will hopefully be easier to ascertain once a more exact notion of the capability has been established. The computers that possess the desirable property of being programmable are also often quite modular in nature: If a certain task is too great, it is not difficult to adjust the size of the apparatus to accommodate the problem. As will be discussed, some types of computers do not appear to lend themselves to easy extension, whereas other types do. It is relevant to understand precisely what the difference between the two types are, as we aim to construct the former type in a biomolecular context.

Recall that we use a nomenclature wherein "Definition" and "Concept" refer to mathematically exact definitions and intuitive, "hand-wavy" definitions, respectively.

3.1 Computers

Concept 3.1 *The word "computer" means any construction, physical or non-physical, electronical or biological, capable of* computing*. That is, capable of performing, in an appropriate sense, a sequence defined by an outsider of internally well-known steps with the purpose of repeated manipulation of some state representation. It is an automaton, but with a possible physical embedding.*

Concept 3.2 *An* execution *of a computer is its performance of a number of well-known internal steps. Thus, the execution is simply the process of performing each small step in a sequence of steps.*

Using these notions, it is evident that a "computer" can be many different things; the desktop PC satisfies the definition, as do theoretical constructions like the λ-calculus [10] and various programming languages. More esoteric constructions like a group of workers equipped with very specific tasks and supervised by a manager can also be said to be a computer, as can a pocket calculator. In the latter case, the computer is quite limited, but there is nothing in the definition disallowing this. The sequence of steps given by an outsider is in this case given at "assembly-time," when the pocket calculator was built in the factory.

3.2 Universal Computers

A universal computer is a computer capable of simulating any other computer of the same type. Also, if a computable reduction from computers of type A exists, it is able to simulate all A-computers. The Church–Turing thesis

states essentially that a universal computer has such a reduction *for all* types A of computers. Universality in that sense thus implies that we will only ever need one computer, if computational speed is not an issue. Alas, this notion of universality by itself is for our purposes a too coarse-grained resolution in which to study computers. A modern-day PC is universal; it can simulate any computer and compute any computable function (within reasonable bounds, as discussed below). Obviously, a single NAND gate cannot achieve this. However, it has been shown that any Turing machine can be simulated using a circuit, and therefore using a number of NAND gates wired together [30, 121]. As our normal PCs are constructed using electronic Boolean circuits, it is hardly surprising that any computer can be constructed using NAND gates. The tempting thing to conclude is then that the NAND gate is universal and equal in expressiveness to a desktop PC. This conclusion is, for the above reasons, not false, but there certainly is a difference, if not in anything else then in practicality, between a PC and a NAND gate! Prompted by this difference, we will introduce the notions of *linguistic universality* and *mechanical universality* in this chapter.

3.2.1 Components

Concept 3.3 *A* component *c is an object with a finite set of input and output channels.* Each output channel has a finite number of* receivers*. The number of receivers on a channel is called* fan-out*. $\mathfrak{M}_i(c)$ and $\mathfrak{M}_o(c)$ denote the input and output* communication methods *of the component c, used in the input and output channels, respectively. Channels have two types:* internal *and* external*. Internal channels connect components with each other, and external channels connect components with the environment.*

Any component needs a mechanism that allows it to communicate with other components. This implies that "some" substrate must be present which can be altered into a number of distinguishable modes by the component, representing the communication method. In order to be able to transmit anything besides a constant signal, there must necessarily be at least two modes. Furthermore, for a computer to be of any practical use, there must be some way that we can input data to it, as well as a way of reading the computed data out again. This is what the external channels are used for, as they allow some "border component" to interface with a user, for instance in the form of a keyboard. The nature of the difference between the two kinds of channels, internal and external, depends, as the channels themselves, heavily on the type of components. Drastically different communication methods may be employed on the two types.

The notion of a component is what we will base our understanding of a "computer" on. A computer is constructed using components and the interplay between them, so in the following it is implicit that a computer is a collection of components put together in some appropriate way (for instance by wiring).

Concept 3.4 *The set CM is the set of all components.*

The presence of CM is of a more philosophical than mathematically exact nature. Our reason for putting it in a disguise of mathematical exactness is the

*Not related to the equivalently named notion of channels found in process calculi such as the π-calculus and κ-calculus.

greatly simplified notation it allows us to use.

Determining which objects are present in CM and which are not is an imprecise action. We are forced to use some common-sense argumentation along the way, as the very definition of what a component is allows us to be very generous with what should be in CM. This generosity must not be exploited to place in CM components which are of a too small or too large level of complexity, ultimately causing the problems we are trying to solve to be hidden in the interior of the components. If it is decided, for instance, that a PC is a component, which is perfectly acceptable according to its definition, the components themselves contain the complexity to perform the actions which we aim to dissect. Likewise, not much use of a component definition with single atoms is likely to be found. Of course, the meaningfulness of a certain component choice must be evaluated in the context of what kind of computer we are trying to build. Taking a PC as a component might make sense if a large scale cluster computer is to be built, and an atom-scale component might make sense if circuits with molecular sizes are sought.

Example 3.1 A NAND gate is a component. It has two input channels and one output channel; the communication methods for input and output are the modulation of electric signals by altering between (ideally) two voltage levels. The physical realization of the channels between components is wires. Two gates can be attached to one output channel of another, making them both the receivers on the same channel. This causes the fan-out of the latter gate to be two. In practical implementations of electronic computers, human-friendly input and output devices have been constructed, such as the keyboard and the monitor. From the user's point of view, this makes the *external* input channels employ the mechanical force applied to keys on a board, and the external output channels employ the colored light on a flat surface.

Example 3.2 Gears are components, having the transferred torque as their method of communication. A gear has one input and one output channel, namely the force applied to the gear and the force applied to other gears as a result. If more than one gear is connected to a certain gear, g, they are all receivers on the output channel of g. External communication with a computer constructed this way could be performed using a set of reserved gears, having special, human-readable markers on them. For instance, a primitive addition function is shown in Figure 3.1, which uses gears and rods as components. Input from a user to the "computer" of Figure 3.1 is given by adjusting the right digit gear, and output is read by observing the left one.

Example 3.3 A DNA oligonucleotide of length n is a component. The capability of the bases A, C, G, and T to form chemical bonds with their Watson–Crick complementary bases embodies both the input and output channels; this property is exploited to pass on and manipulate information in a variety of the DNA-based computing schemes described in Chapter 2. An internal input and output communication method could for instance be DNA polymerase, as discussed in Chapter 2. One example of an external communication method for reading out computational results is the surface-based approach, outlined in Chapter 2, in which DNA oligonucleotides are attached to small areas on a gold surface, which can be observed as changes in fluorescence intensity [71].

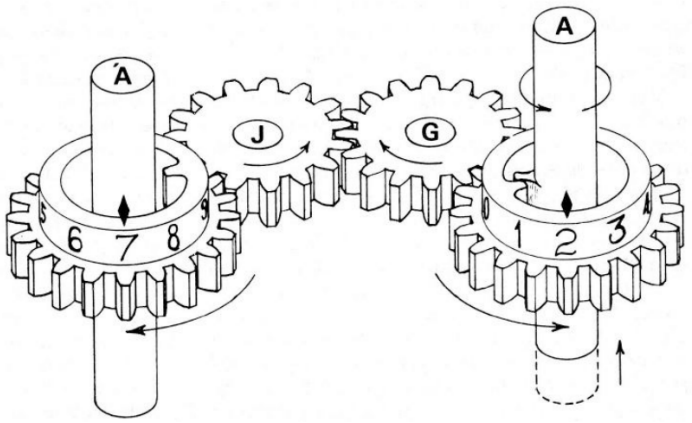


Figure 3.1: *Gears as components in a simple computer. The gears on the drawing implement addition modulo 10 of single digit numbers. Turning the axis A such that the right digit gear shows some number, lifting it such that the right gear connects with the left through gears J and G, and turning A again, causes the digit to be added modulo 10 to the number stored on the left digit gear. The illustration is taken from [26].*

3.2.2 Computers as Component Structures

A computer always simulates at least one Turing machine. The word "simulate" in this use is an analytical construction; an actual computer does not explicitly simulate anything, it does what it is supposed to, like modifying bits in some CPU. However, according to the Church–Turing thesis, we know that there exists *some* Turing machine capable of doing the same work given enough time and space, and hence we say that the computer "simulates" that particular Turing machine. A practical computer is able to simulate more than one Turing machine, but even if the computer is a hardwired small circuit, it simulates some Turing machine. In that case, the simulated Turing machine is the machine that computes the Boolean function of the circuit.

Concept 3.5 *Let m be a computer simulating the Turing machine t and let k be the number of tape cells written in t at a certain point in time, where $|t|$ is the size of the description of t. Let $n = k \cdot \log |t| \cdot \log k$; by the* configuration *of a computer we mean the n-dimensional vector c such that the Turing machine's current tape contents, its internal state, and the position of the reading head on the tape is stored in c.*

The notion of a configuration of a computer does not take external properties of the computer and its environment into account. Neither the temperature nor the weight of the computer is part of the configuration, unless the computer is constructed in such a way that these are internal properties. Building a computer using components that communicate in some way using temperature changes, for instance, would result in the temperature being part of the configuration, and the construction of a computer from components that communicate by altering their mass would cause the mass to be part of the configuration.

Example 3.4 A configuration of an ordinary PC is the memory contents, the hard drive settings, the register values, etc.

Example 3.5 A configuration of a computer constructed using gears is the positions of the gears relative to each other.

Example 3.6 One type of configuration of a DNA-based computer is the sequence of base pairs coding for states and symbols of a finite automaton used by Benenson et al. in their DNA-based DFA described in Chapter 2 [17].

To be able to speak about computers in general, we need some category from which we can take the objects of our discussion. This category, the set of all computers, consists of tuples which each couples a computer with a configuration of it. Hence, one physical computer is present several times in the category, once for each configuration.

Concept 3.6 *The set M:*

$$M = \big\{(m, c) \;\big|\; \textit{for all computers } m \textit{ and all its configurations } c\big\}$$

is the set of all computers in every possible configuration.

Remark 3.1 The distinction between a computer and its configuration is explicit in the tuple notation of Concept 3.6. A configuration of a computer is not a part of the computer itself. For a Turing machine this means that while the raw tape is part of the computer, the contents of the tape is not. Likewise is the state of the Turing machine not part of the computer, even though it is the computer that possesses the ability to be in more than one state.

As with the set CM of all components, the presence of the set M is a somewhat wavy statement, as it may contain extremely diverse objects. Common sense need be employed when we discuss computers and their inclusion in M, so as not to categorize, e.g., a beach ball as a computer, capable of "computing" exactly the trajectory of a beach ball of that particular size and mass when thrown. Again, our common sense must take the context into account, as there might be scenarios in which it is in fact beneficial to study the "ball-as-a-computer" model.

We shall discuss computers from a physical perspective, taking into account the actual, physical construction of the computer when assessing its capabilities. To make it easier to do this in a mathematical notation, we introduce an abstraction over the physical layout of the way the components, from which the computer has been constructed, are put together.

Concept 3.7 $\mathfrak{A}\colon M \to (V, E)$ *denotes the* assembly graph *of a computer.* $\mathfrak{A}(m)$ *is the graph $G = (V, E)$ such that (i) each vertex in V corresponds to one component in m and (ii) each edge in E corresponds either to an internal channel, connecting two components, or to an external channel.*

An assembly graph can be thought of as a generalized form of circuit diagram: A circuit diagram is an assembly graph of a computer built with Boolean logic gates, etc., as components. Hence Example 3.7.

The nodes and edges in an assembly graph contain state information. In an assembly graph for an electrical circuit, each node and edge corresponds to logical components and wires, respectively. Both the wires and the logical

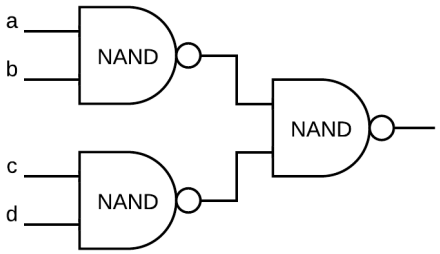


Figure 3.2: *The assembly graph of the circuit from Example 3.7. Not shown on the drawing is the contents of the nodes and edges, i.e., the electrical charge on the wires that makes up the configuration of this particular computer.*

components can contain electrical charge, and so a particular configuration of a circuit is represented in the assembly graph by a "payload" of electrical charge on the nodes and edges. Likewise, an assembly graph for a Turing machine has as one of its components a single "cell" on the tape of the Turing machine, i.e., the tape itself can be construed as consisting of several cells attached to each other. These cell-components carry with them a piece of state information, namely the symbol currently stored at the position on the tape corresponding to the cell-component.

Example 3.7 The very simple computer implementing the Boolean function

$$f = \neg\big(\neg\,(a \wedge b) \wedge \neg\,(c \wedge d)\,\big)$$

straightforwardly using NAND gates has an assembly graph with three vertices and seven edges, see Figure 3.2. Four of the edges are external "input" edges; two edges are internal, connecting NAND gates; one edge is external, serving as the output of the function. The edges contain information about the electrical charge on the corresponding wires. The state of the computer is entirely described with what is on the wires, as no explicit memory is present in the system.

Definition 3.1 *The set RE is the set containing all recursively enumerable languages, represented as encodings of Turing machines that recognize them.*

In the rest of this Chapter, whenever the "language" p is mentioned, an encoding of a Turing machine recognizing p is meant.

For notational convenience, we introduce the *recognizer function.* As it is undecidable, it can be regarded as a mathematical shorthand for the English statement "the computer recognizes the language."

Definition 3.2 *Let* $R\colon\ M \times RE \to \{\mathsf{T}, \mathsf{F}\}$ *be the* recognizer function *defined by:*

$$R(m,l) = \begin{cases} \mathsf{T} & \text{if the computer } m \text{ recognizes the language of } l; \\ \mathsf{F} & \text{otherwise.} \end{cases}$$

Remark 3.2 The recognizer function is undecidable. If R were decidable, we could construct a Turing machine computing it, enabling us to construct another Turing machine solving the Halting problem: On input (m, x), run the

Turing machine for $R(m, x)$ and return what it returns. This machine always halts if R is decidable, and hence R is undecidable.

We need some method to bound infinite languages to subsets containing only strings shorter than a certain length k. This enables us to consider computers with a finite amount of memory that cannot recognize strings larger than k.

Definition 3.3 *Let* $I_k \colon RE \to RE$, $k \in \mathbb{N}$ *defined by:*

$$I_k(p) = \left\{ p' \;\middle|\; p' \subseteq p \wedge \forall s \in p' : |s| \leq k \right\}$$

be the function that takes a Turing machine recognizing a language and makes a Turing machine recognizing strings in the language that are shorter than k.

Remark 3.3 For every k, the I_k-function is computable: It is possible to construct a Turing machine that both performs a simulation of another Turing machine and performs a length check.

3.2.3 Assembling Computers

Concept 3.8 *Let* $\phi_k \colon \mathcal{P}(CM) \times RE \to M$, $k \in \mathbb{N}$ *denote the* assembler function *that, given a set of components* C *and a Turing machine recognizing the language* l*, returns a computer composed of components in* c *recognizing the finite language* $I_k(l)$.

Our notion of an assembler function presumes nothing about the computability or realizability of assembling a computer. However, as will become evident, we limit ourselves to computable and "realistic" assembler functions, as neither the uncomputable functions nor the physically unrealizable ones carry any other significance for a practical construction of a computer than mere artistic novelty.

Remark 3.4 Let the projections $\pi_i \colon T^n \to T$, $1 \leq i \leq n$, be the functions that each returns the ith element of an n-tuple. For two languages l and l', the relation

$$\pi_1\big(\phi_k(c, l)\big) = \pi_1\big(\phi_k(c, l')\big)$$

does *not* hold in general, because a new computer is assembled for each language.

Example 3.8 (ϕ_k^{NAND}) An assembler function for NAND gates is a function that takes the singleton set containing a NAND gate, along with a Turing machine description t that recognizes some language l. It physically wires together a number of NAND gates such that they recognize a finite subset $I_k(l)$ of the language l, for instance using a method similar to the one described by Chazelle [30] and Wegener [121].

Example 3.9 (ϕ_k^{gears}) For gears, one example of an assembler function is a function that takes a set with different types of gears, rods, etc., and a Turing machine recognizing the language l. It assembles the gears into discrete arithmetic units, performing basic multiplication and addition operations on a set of "integer gears," representing the Turing machine state and -tape (see Figure 3.4.) To indicate whether a given input is in the language $I_k(l)$ or not (that is, we do not know it yet), a special, "acceptance module" is added, that reacts if the integer representation of the state reaches a special value, corresponding to a halting state of the Turing machine.

Example 3.10 (ϕ_k^{DNA}) Assembling the DNA components from Example 3.3 to recognize a language $I_k(l)$ could be done in a number of different ways, as described in Chapter 2. One assembly function is the now-classic Adleman–Lipton method. All solution candidates are enumerated in a DNA representation, causing the "execution" of the computer to be a sequence of ligation reactions and laboratory washes. Input is given at assembly time, coded in the solution candidate DNA strands, and output is obtained by gel electrophoresis.

As the usefulness of an assembler function also depends on its ability to create simple structures that are easily extended, we need some method to distinguish assembler functions from each other by this quality. We measure the *size* of the computer, and the size change caused by constructing the computer to a larger problem.

Definition 3.4 *Let* $\|\cdot\| : M \to \mathbb{N}$ *represent a notion of* computer size.

Example 3.11 Several size notions of a computer based on Boolean logic gates could be used: the number of components, the longest path through the circuit composed of the gates, etc., [121, Definition 3.2].

Example 3.12 A notion of size for a mechanical computer constructed using a set of gears is either the total number of gears used, or some function of the assembly graph indicating the amount of power required to turn the gears (assuming a significant energy loss in the transmission of torque between gears).

Example 3.13 One possible size notion for a DNA-based computer is the number of DNA oligonucleotides that must be created in order for the computer to work, as this necessitates laboratory steps that must be performed by a human. The Adleman–Lipton method [3, 70], for instance, requires the synthesis of all possible solutions to a problem prior to the computation phase. The number of possible solutions, and therefore the number of necessary DNA strands, is a notion of size for that particular computer.

Example 3.14 If the amount of necessary laboratory steps does not only depend on the number of possible solutions, the size notion from Example 3.13 can be altered to just refer to the total amount of "laboratory work," defined, e.g., as the amount of man-hours required of the laboratory assistant. As mentioned in Chapter 2, Adleman notes that the amount of laboratory steps required to operate his DNA computer should grow linearly with the problem size [3]. However, as also noted, the required work in the laboratory was about seven days, representing a rather high man-hour cost for the computation of a Hamiltonian path on a seven-node graph.

3.2.4 Programming Computers

Concept 3.9 *Let* $\psi_k \colon M \times RE \to M$, $k \in \mathbb{N}$, *denote the* programming function. *Given a computer,* m, *and a Turing machine description of a language,* l, *the function modifies the configuration of* m *such that it recognizes the finite language* $I_k(l)$*:*

$$\psi_k\big((m, c), l\big) = (m, c').$$

The need for a bounded programming function, as opposed to an unbounded one capable of recognizing arbitrarily large languages, arises from the fact that

we wish to develop a theoretical foundation for *buildable* computers. We cannot build an infinitely large computer, whence we cannot expect it to possess an infinitude of configurations. Still, key to the usefulness of computers is their ability to easily or cheaply improve the finite approximation to the notion of an infinitely large computing device (e.g., a Turing machine with infinite tape).

The programming function does neither rebuild nor reassemble the computer. The internal configuration is changed, thereby causing the entire computer to behave differently, essentially simulating another computer. As the actual, physical realization of a configuration varies with the type of computer, the method used to modify it is necessarily vaguely described. As examples, the configuration of a Turing machine is altered by manipulating the tape contents; the configuration of an electronic computer is manipulated by setting bits in the memory on and off; the configuration of a mechanical computer is changed by turning some special gears and providing some punched cards, and so on.

Example 3.15 "Programming" in the sense of Concept 3.9 is broadly described, as it entails that the computer consisting of a single NAND gate is programmable too, albeit in a very limited sense: Fixing one of the inputs of the gate to "true" causes the NAND gate to simulate a NOT gate. Equivalently, the identity gate can be "programmed" into the NAND gate. This style of programming uses the fact that the configuration of a NAND gate is entirely described by the electricity levels on the incoming wires.

Example 3.16 There exists a computable function that maps any Turing machine with an encoding shorter than n to a configuration in a PC (with at least $O(\log n)$ bytes of memory). This function alters the configuration of the PC by loading machine instructions into its memory. Therefore, a PC has a programming function, which is a compiler from some Turing machine description to the machine code of the PC's CPU.

Example 3.17 A gear computer capable of reading and writing punched cards while also executing instructions encoded on the cards has a programming function, provided that the instruction set is of an adequate size. If this requirement is met, the programming function reads a Turing machine description and perform a translation of it into a sequence of instructions, by punching holes in the cardboard pieces in the fashion prespecified by the computer. The tape of the Turing machine is simulated by instructions that cause the machine to read or write on auxiliary cards. An injective mapping from the alphabet Σ_{TM} of the Turing machine to the alphabet Σ_{Holes} of the gear computer is contained in the programming function.

The assembler function ϕ_k and programming function ψ_k are akin, as both produce some computer recognizing some language. The crucial difference is that the assembler function utilizes a number of components, potentially several of the same kind, and "wires" them together, whereas the programming function performs a permutation of some internal parts of a configuration in a computer, thus not causing the assembly graph $\mathfrak{A}(m)$ of m to be changed. The contents of the nodes and edges of $\mathfrak{A}(m)$ encoding the configuration is changed by the programming function, but the graph structure remains untouched.

3.2.5 Mechanical and Linguistic Universality

Definition 3.5 *Let* $e\colon V \times E \to \{0,1\}$ *denote the* computable *graph encoding which represents the finite graph* $G = (V, E)$ *as a list of* $|E|$ *tuples with adjacent nodes, coding each node* i *with a sequence of symbols* $s_i \in \{0,1\}^{\log_2 |V|}$.

When we use the notion of an encoding of a graph in the following, we shall carelessly pretend to assume an efficient encoding to be present. "Encoding" of a graph, as it is used later, therefore implicitly refers to Definition 3.5.

Definition 3.6 *Let* $n \in \{\mathbb{N} \cup \{\omega\}\}$. *Let the family of assembler functions denoted by*

$$\Phi_n = \left\{\phi_k \mid k < n \wedge \phi_k \in \mathcal{P}(CM) \times RE \to M\right\}$$

be defined by:

(i) There exists a Turing machine computing the function:

$$\tau_T\colon \mathbb{N} \to \mathcal{P}(CM) \to RE \to \big(\mathcal{P}(CM) \times RE \to M\big)$$

that computes $\phi_k(C, p)$ *given* k, *a set of components* C, *and a language* p.

(ii) The construction of the computer described by the Turing machine from (i) is physically realizable, *i.e., there must exist some device capable of performing the steps described by the machine.*

Then Φ_n *contains* computable *assembler functions.*

The definition of Φ_n operates with "some device" that can be attached to a Turing machine such that the former performs the actions encoded by the latter. A suitable way to grasp an intuition about this is a robotic arm attached to a Turing machine which encode a sequence of steps that the arm must perform. The "device" need not be a robotic arm, but the possibility of automation must be present. The family Φ_n enables us to exclude unrealistic assembler functions, such as those employing magic, from our further considerations.

Definition 3.7 *Let* $n \in \{\mathbb{N} \cup \{\omega\}\}$. *Let the family of programming functions denoted by*

$$\Psi_n = \left\{\psi_k \mid k < n \wedge \psi_k \in M \times RE \to M\right\}$$

be defined by the Turing machine L *computing the function:*

$$\tau_L\colon \mathbb{N} \to M \to RE \to \big(M \times RE \to M\big).$$

Given k, *a computer* m, *and a language* p, *the Turing machine* L *computes* ψ_k. *Then the family* Ψ_n *contains* computable *programming functions.*

The presence of Ψ_n implies that a compiler can be constructed, as the Turing machine L computes a representation of a language l in the "language" of configurations of computers, essentially performing an automated translation from one representation of a language to another.

Definition 3.8 (Mechanical Universality) $C \in \mathcal{P}(CM)$ *is* mechanically universal *with* strength z *if:*

$$\forall p \in RE\colon\ \forall k < z\colon\ \exists \phi_k \in \Phi_z\colon\ R\big(\phi_k\left(C, p\right), I_k(p)\big).$$

Definition 3.9 (Linguistic Universality) $m \in M$ *is* linguistically universal *with* strength z *if:*

$$\forall p \in RE\colon\ \forall k < z\colon\ \exists \psi_k \in \Psi_z\colon\ R\big(\psi_k\left(m, p\right), I_k(p)\big).$$

Remark 3.5 In the definitions of the two types of universality, we apparently just pick an assembler or programming function out of thin air. Luckily, this is not the case, as we pick them from the two sets defined by their very computability in Definitions 3.6 and 3.7, ensuring that a linguistically universal computer cannot rely on some uncomputable programming function. Otherwise it would be impossible to program as no compiler can be constructed, and therefore be worthless for practical applications.

A linguistically universal computer is able to alter its behavior based on its input — a program written in a programming language. Given this input, the computer performs an execution of the specialized computer described in the language, hopefully producing the desired result. Thus, we can formulate the "path" from problem to solution as:

$$\text{problem} \rightarrow \text{formulation in a suitable language} \rightarrow \text{execution} \rightarrow \text{result}.$$

A mechanically universal computer is unable to alter its behavior in the same way. However, for any specific computable problem instance we can always construct a new computer that solves it. It is therefore necessary to assemble it anew for each problem instance, if the behavior of a linguistically universal computer is to be reproduced in a mechanically universal one. The abstract path from problem to solution can then be formulated as:

$$\text{problem} \rightarrow \text{assemble computer and execute} \rightarrow \text{result}.$$

The execution step in this abstraction is implicit in the assembly of the computer, as it cannot perform anything else once it is assembled. Therefore, the computer cannot be said to "execute" in the same sense as the linguistically universal computer, as it simply does what it was built to do, and not what the computer described by an outsider was supposed to do. The simulation becomes a physical realization of the simulated.

Mechanical universality can be "hidden" if it is contained in another, linguistically universal computer. For instance, if a programmable computer capable of assembling electronic circuits with NAND gates according to a software specification were present, the "mechanicalness" of the universality of the NAND gates would be hidden from our point of view. This "assembly" can also be likened to the compilation of one programming language to another; the transformation of one representation of a Turing machine into another.

A linguistically universal computer is constructed using a set of components. The next Lemma shows that linguistic universality is "stronger" than mechanical universality, in the sense that the latter is included in the former.

Lemma 3.1 If a computer m consisting of a finite number of components is linguistically universal with strength $n \in \mathbb{N}$, the set of components C from which m has been constructed is mechanically universal with strength (at least) n.

Proof. Let $A = \{\mathfrak{A}(x) \mid x \in M\}$ be the set of all assembly graphs. Construct the Turing machine G that computes the function:

$$\tau_G \colon A \to RE \to \mathbb{N} \to \mathcal{P}(CM) \to RE \to \big(\mathcal{P}(CM) \times RE \to M\big)$$

1. Take as input (i) an assembly graph $\mathfrak{A}(x)$, (ii) a Turing machine G' computing τ_L of Definition 3.7, (iii) an integer k, (iv) a set of components C, and (v) a language p.
2. Run G' on k, x, and p to obtain the new configuration d' of x.
3. Fix the assembly graph of x such that it contains the configuration d'.
4. Write the newly created assembly graph using an encoding understandable by a "device" in the sense of Definition 3.6 (ii).

The finiteness of the computer m is not changed when altering m's configuration. Because m is linguistically universal, there exists a Turing machine L that computes its programming function. Running the Turing machine G with the component set argument fixed to the component set of m, the assembly graph fixed to $\mathfrak{A}(m)$, and the Turing machine argument fixed to L, we obtain the Turing machine T of Definition 3.6. This implies that Φ_n exists, and we have:

$$R\big((m,d), I_k(p)\big) \implies \exists \phi_k \in \Phi_n \colon R\big(\phi_k(C,p), I_k(p)\big),$$

coupling the linguistic universality of m to the mechanical universality of C:

$$\begin{aligned} &\forall p \in RE \colon \forall k < n \colon \exists \psi_k \in \Psi_n \colon R\big(\psi_k\,(m,p)\,, I_k(p)\big) \\ \implies &\forall p \in RE \colon \forall k < n \colon \exists d \colon R\big((m,d), I_k((p))\big) \\ \implies &\forall p \in RE \colon \forall k < n \colon \exists \phi_k \in \Phi_n \colon R\big(\phi_k(C,p), I_k(p)\big). \end{aligned}$$

□

Remark 3.6 Lemma 3.1 only considers finite computers. However, for a linguistically universal computer m with infinite strength ω, the Lemma implies that every finite approximation m' to m having strength $n \in \mathbb{N}$ induces a mechanically universal counterpart in C, the components of m'.

Remark 3.7 The reverse implication of Lemma 3.1 does not hold: For a linguistic computer to be realizable, a way to encode foreign Turing machines in the computer must be present. Mechanical universality per se does not imply that this is possible.

Example 3.18 The singleton set that contains the NAND gate is mechanically universal. The assembly functions ϕ_k could be ϕ_k^{NAND} from Example 3.8. The strength of the mechanically universal computer is infinite (ω), as arbitrarily large circuits can be constructed.

Example 3.19 The set containing gears of all sizes is mechanically universal: As it is possible to construct mechanical calculators capable of basic arithmetic, such as Blaise Pascal's *Pascaline*, it is possible to define the assembly functions ϕ_k to assemble a calculator that simulates the operations of the given Turing machine using a suitable Gödel-numbering, as outlined in Example 3.9 and illustrated in Figure 3.4. The component set has infinite strength, as arbitrarily large calculators can (in principle) be constructed.

Example 3.20 The classic, theoretical Turing machine with an infinite amount of tape also has an infinite amount of configurations, as each arrangement of the tape cells represents a particular configuration. The machine has programming functions ψ_k, because any Turing machine can be encoded on the tape. Thus, the machine is linguistically universal. Because it has an infinite amount of tape there is no upper bound on the size of Turing machines that can be encoded, meaning that the strength of the machine is infinite.

Example 3.21 When constructing an actual, mechanical Turing machine, we cannot take the liberty of having an infinite amount of tape. On such a computer there is only a finite amount of Turing machines encodeable with a programming function, resulting in a strength $n \in \mathbb{N}$.

Example 3.22 A PC is linguistically universal because we can construct a mapping between any Turing machine that recognizes a language and the machine language of the CPU. The strength of the computer depends on the amount of RAM, but it is finite.

Example 3.23 Charles Babbage's *Analytical Engine* is a linguistically universal, mechanical computer. The programming functions ψ_k are mappings between Turing machine descriptions and punched cards. As with the PC, the strength is a measure of the memory of the computer, which in this case is stored on axles with gears maintaining certain positions.

Example 3.24 Immortal human beings are linguistically universal computers: The programming functions ψ_k are constructed such that they equip a person with detailed instructions in how to perform the actions of the given Turing machine. Without the immortality requirement there would exist languages requiring more time than the life span of the human to compute, rendering the computer non-universal.

Example 3.25 As mentioned in Chapter 2, Su and Smith created a NOR gate using DNA strands [109]. This construction represents a mechanically universal computer, as there exists an assembler function that represents any Turing machine as a circuit composed of NOR gates (simply alter ϕ_k^{NAND} to use NOR gates, for instance).

3.2.6 Computers with Localized and Distributed Control

To quantify what it is that makes some computers easier to "extend" than others, thus severely improving their usefulness as they can be applied to a broader range of problems, we will introduce a distinction between two types of computers. The reader is reminded about the notion of a *cut* in graph theory:

Definition 3.10 *A* cut *of a graph $G = (V, E)$ is a partition, $C = (L, R)$, of the vertices of G such that $L \cup R = V$ and $L \cap R = \emptyset$.*

Definition 3.11 *A computer m_k with strength k has* localized control *if there exists a cut $C = (L_{m_k}, T_{m_k})$ of the assembly graph $\mathfrak{A}(m)$ such that:*

(i) L_{m_k} represents all the components in m_k that implement the functionality of the control mechanism of t;

(ii) T_{m_k} represents all the components in m_k implementing the tape of t;

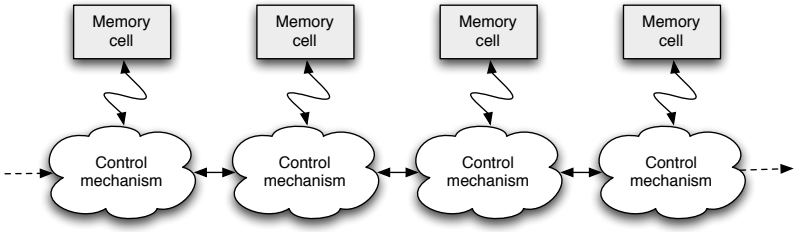


Figure 3.3: *Schematic view of the simulation suggested in [30] using electronic Boolean circuits to simulate the behavior of any Turing machine. The "control mechanism" on the diagram is a circuit implementing the program of the Turing machine, and each "memory cell" on the diagram is another circuit corresponding to one cell on the Turing machine's tape.*

(iii) T_{m_k} *and its component structure can be increased in size to* T_{m_n}, $n > k$, *with* L_{m_k} *unaltered.*

Definition 3.12 *A computer that does not have localized control has* distributed control.

A computer with localized control possesses the ability to be increased in strength by simply "adding more tape" — in the appropriate sense. A computer with distributed control cannot have its strength increased in the same way, as there is no "tape" to be extended. The price of increasing the strength of a computer with localized control is thus in principle smaller than that of increasing the strength of a computer with distributed control. The distinction presumes that the repeated addition of the control mechanism is cumbersome and expensive to do. This need not be the case; a distributed network of PCs is easily extendable with new PCs, even though the exact location of the "tape" is unclear.

Example 3.26 Chazelle's Turing machine simulation used in Examples 3.18 and 3.8 is a computer with distributed control: There is no way to make a cut that satisfies the criteria in Definition 3.11, as the technique used in the simulation does exactly the opposite — it supplies a copy of the entire control mechanism each time a new piece of memory is required, see Figure 3.3 [30].

Example 3.27 The computers constructed using the mechanically universal gear component set from Example 3.19 have distributed control, as memory cannot be added to a completed construction. A sketch of a computer constructed in the way outlined in the example is shown in Figure 3.4. It is evident from the illustration that the addition of more tape would necessitate each "arithmetic unit" to be modified.

Example 3.28 The Turing machines from Examples 3.20 and 3.21 do both trivially have localized control: As can be seen on the illustration on Figure 3.5, tape can be added without affecting the rest of the computer. For the infinitely large Turing machine it would not make sense to enlarge it, but the property does allow us to construct ever more "complete" finite approximations to the, from

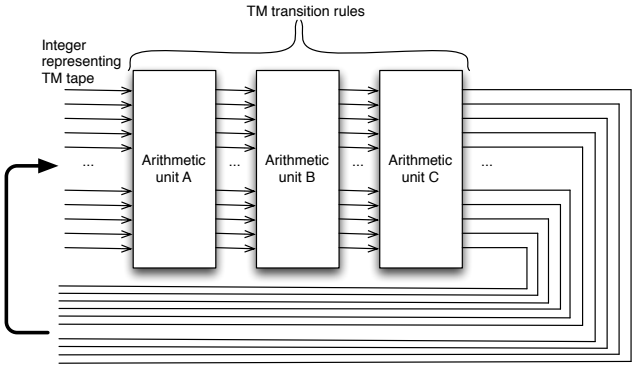


Figure 3.4: *Sketch of a computer constructed using the method mentioned in Example 3.19. Each transition rule is implemented as a sequence of arithmetic operations, and the state and memory of the computer is encoded in one large integer, represented as a set of lines on the illustration. Each arithmetic unit operates on the integer and possibly changes the state. As the state is encoded as part of the integer, it can be changed arithmetically, and likewise for the tape contents. The integer runs in a loop, to allow the computer to loop. A special "halting state" checker could be placed in one end, responding only if the state part of the integer represents a halting state. Clearly, we cannot enlarge the integer that the computer works upon without enlarging each "arithmetic unit," representing the control unit of the Turing machine.*

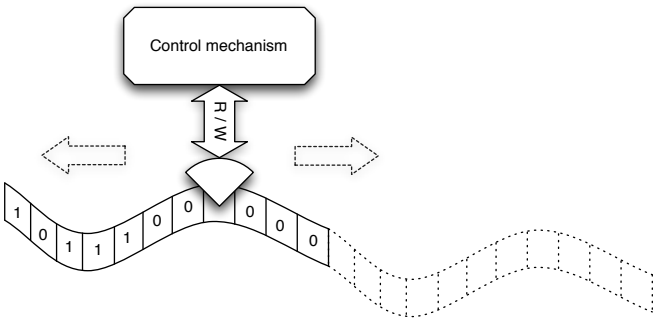


Figure 3.5: *A stylized Turing machine. Tape can be added without the manipulation of the components making up the control mechanism.*

an engineering point of view, impossible construction that is a truly universal (i.e., with an infinite amount of tape) Turing machine.

Example 3.29 The PC from Example 3.22 has localized control, as it is possible to add memory to the machine without altering the CPU.

Example 3.30 The Analytical Engine from Example 3.23 has localized control, as it possesses the ability to write punched cards, offering the possibility of the extension of the innate "memory axes" with punched card tape, at the expense of increased computation time [26].

Example 3.31 As we have observed that immortal humans can be viewed as linguistically universal computers in Example 3.24, we observe that, if more memory is needed, pen and paper can be easily inserted into the system, thereby heightening the strength. Hence, it is a computer with localized control.

Example 3.32 The DNA-circuit computer from Example 3.25 has, for the same reasons as its electronic counterpart in Example 3.18, distributed control.

3.3 Challenges of Biological Universality

Whenever authors of biological computing articles claim to have constructed a universal computer, it must be taken with a grain of salt. The constructions are mechanically universal computers, as the several groups constructing biological Boolean circuits demonstrate, but not linguistically universal computers: We cannot take Stojanovic's and Stefanovic's tic-tac-toe circuit and ask it to do word processing or database searching. Not even tic-tac-toe on a four-times-four board is possible without the effort of rebuilding the circuit [107]. Likewise, the numerous solutions to instances of NP-complete problems using biological computing techniques call for the same care in the use of the universality-attribute. The constructions are tailor-made devices that do one specific thing, in these cases solving (small) instances of NP-complete problems. We have no automatic method to take any algorithm and pour into, say, Braich et al.'s construction that solves the 20-variable instance of the 3-SAT problem [23].

Naturally, the construction of a linguistically universal biomolecular computer poses several challenges, of which a few are enumerated here.

Signal propagation. In conventional electronic computers, the signals between components, i.e., Boolean logic gates, flow in discrete "lanes," ensuring that a signal from component A to component B does not accidentally end up in component C. This property allows the computer to use the "same" signal for any communication, namely high/low voltage. If the components are implemented as free-flowing molecules in a liquid or gel, for instance, this discreteness disappears. To ensure that signals end up where they are supposed to be, we are forced to implement the computer such that it uses different signals for different pathways through it.

Error tolerance. Intuitively, a high error rate is expected from a biological computer. This intuition comes both from the issue with the signal propagation, leading one to believe that sometimes some signals end in the wrong place, but also from observing existing biological systems, such as animals. If these systems were engineered, they would be so with a fundamentally different philosophy than our usual engineering philosophy — "our" machines tend to be built with high precision in mind, making the more precise machine the better one, as it benefits from being able to make its gears run smoother, its bits less faulty etc. This is effective, but it also has weaknesses: In a computer, a single bit flip in the wrong place can cause it to crash, and in a car, one broken gear out of several hundreds can be fatal. Natural systems appear to have a more plastic mode of working. A few dead cells in an organism is nothing to worry about; they will just be replaced. A biological computer will most likely need this robustness in order to function "amortizedly" correct.

Input and output techniques. Some way to read out the computed data must be present if the computer should be of any practical use. Likewise, a method for input is needed if a computer capable of executing software is desired. If the computer should also be *practical*, these steps should be "fast."

Memory I/O. Writing and reading contents to and from memory must be made possible in some way, if the computer is supposed to be linguistically universal.

Energy consumption. As computation is a physical act, energy is required for the execution of a computer. Failure to provide this energy in some external form, for instance as ATP molecules, and instead relying on some internal properties of the chosen implementation substrate will cause the computer to be a construction with a short lifespan, as it would stop working when the internal energy source dried out. One could imagine some cryptographic benefits from this, however, as it allows creation of self-destructing software and hardware.

4 The Blob Model

In [54] the authors aim to design a model for a biomolecular computer that is *naturally programmable* and *biologically feasible*. In order to fulfill the latter criterion, a set of "natural" constraints for the model were formulated:

No data pointers. An instruction and the data must be physically adjacent using, e.g., a chemical bond. *Pointers* in the sense of a traditional programming language, i.e., an address of a part of the memory that can be looked up, does not exist. What is colloquially referred to as a "pointer" is a physical bond.

No action at a distance. If an instruction has to alter data that is situated far away, it must be through a chain of local actions. This is caused by the constraint that no real data pointers exist, making it impossible to "point" at data arbitrary places and manipulate it.

Control flow cannot be arbitrarily changed. As with the data manipulation, control flow change must happen locally, i.e., there cannot be any equivalent to C's `goto`.

Furthermore, programs must be data and vice versa if programmability in the same sense as on a normal PC is desired. If the two things were different, there would not be any way to make, say, a self-interpreter purely programmatically.

As discussed in [117], the constraints presented here make programming in the blob model fundamentally different from that of classical programming languages running on classical hardware (e.g., Python on a PC).

4.1 Blobs

The *blob model* is a model for biomolecular computation suggested in [54] with the constraints mentioned above in mind. A "computer" in this model is a "soup" of biological molecules, referred to as "blobs," connected to each other through local bonds, for example chemical bonds. The atoms of the computer are the blobs, which form the basis of both the software, hardware, and data.

Definition 4.1 *A* blob *is an object with four* bond sites*, and a* cargo *of eight bits. The bond sites are numbered "0," "1," "2," and "3."*

Each bond site may be connected to another blob or be unused. Figure 4.1 shows a blob with its cargo and four connection sites. Several blobs can be linked together through the bond sites, shown on Figure 4.2a. Using its eight cargo bits, a blob either encodes an instruction or carries data, referred to as a *program blob* or a *data blob*, respectively. A program is thus a sequence of program blobs, connected using chemical bonds on the bond sites. Data is likewise a sequence of data-carrying blobs connected on the bond sites. Figures 4.2a and 4.2b illustrate program and data blobs; the two leftmost blobs are program blobs and the two rightmost ones are data blobs on both illustrations.

Real programs consist of a set of program blobs connected in an appropriate way (the actual program) and a set of data blobs (the memory, input, and

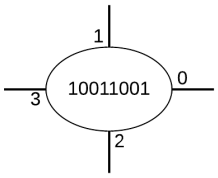


Figure 4.1: *A blob with the payload* 10011001*.*

output). The program operates on the memory, reacting based on the currently active instruction and the currently active memory location. Thus there are always one program blob and one data blob that "communicate," and it is only this program blob and data blob that is executed or read/written, respectively. This is a design decision, most likely taken because it would make the construction simpler and because it makes the reasoning about the system more clear. The most obvious drawback with it is that the "natural parallelism" inherent in the "soup" of blobs is forced into a sequential behavior. The blob model contains the following well-formedness criterion:

Definition 4.2 (Well-formedness criterion.) *At any point in time, exactly one program blob and exactly one data blob are connected at bond site "0." These are referred to as the* active program blob *and* active data blob*, or* APB *and* ADB*, respectively. The bond connecting the two is called* ***.***

Figures 4.2a and 4.2b show the program and data blobs, connected by the bond site *.

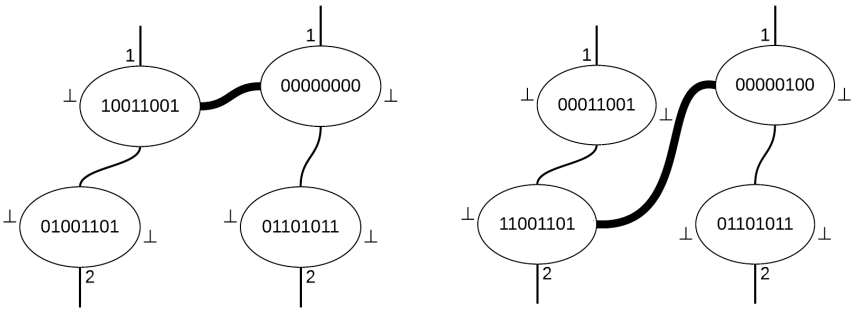
4.2 Program Blobs

The eight bits in the cargo of the program blobs are arranged as follows:

- The first, high-order bit is the *activation bit*, indicating that the blob is the APB, i.e., "1" means that it is the currently executed instruction.
- The seven other bits are used as instructions, making room for a total of $2^7 = 128$ instructions. Each instruction is divided into an operation code and between zero and two arguments, analogous to conventional assembly code.

The seemingly arbitrary choice of the number of bond sites on a blob is justified in [54], using the argument that one bond site is needed to connect a program blob with its predecessor, another one is needed to connect the program blob to a data blob when they are the APB and ADB, and another two are needed to allow for branching operations. In other words, maintaining the program blobs in a linked sequence, while also allowing for branch operations, requires at least four bond sites. The branch operations are necessary in order to simulate `if`-statements and other conditionals, such as `while`-loops.

Likewise, the cargo size of the blobs is also purely a design decision, based on an estimate that eight bits leave enough room for an adequate amount of instructions, while also being practical for containing numbers. The design decision is volatile, meaning that, given practical reasons, it would be easy to alter the model to contain, e.g., ten bits per blob.



(a) *Program and data blobs connected at the bond site* ***** *(the fat line).*

(b) *Program and data blobs connected at the bond site* *******, *one execution step later.*

Figure 4.2: *Execution of an* `SCG` *instruction.*

Programs consist of a sequence of program blobs, and are thus by nature sequential. They need not be linear, as special "conditional branch" instructions may choose between two successor bond sites. For instructions that cannot change program flow, the successor bond site is always "2," which can be seen on Figure 4.2. Thus, a blob is connected through its bond site "2" to its successor's bond site "1," or analogously: a blob is connected to its predecessor on its bond site "1." Bond site "0" is used to connect the APB and ADB, by Definition 4.2, and bond site "3" acts as a secondary successor site, used by branch operations.

The instruction set of the blob model contains 13 operation codes that either manipulate a data blob, branch in the sequence of program blobs, make a fan-in, "create" an entirely new data blob, or cause the program to halt. They are grouped as follows:

1. The manipulation of data cargo bits (`SCG`).
2. Branch operations (`JCG`, `JB`).
3. "Insert," the creation and insertion of new data blobs into the data blob sequence (`INS`).
4. Manipulation of bond sites on the data blobs, altering the sequence of data blobs (`SWL`, `SBS`, `SWP1`, `SWP3`, `JN`, `CHD`).
5. A special fan-in instruction (`FIN`).
6. A special "destination bond site" instruction (`DBS`).
7. A special halt instruction (`EXT`).

As any blob can only have a limited amount of "pointers," i.e., bonds, from other blobs, it is not possible to have, for instance, one point in a program to which program flow can jump from several (more than three) different places. This is the reason for the fan-in blob, as the effect of unbounded connections to a certain program blob b can be obtained by placing several of the fan-in blobs before b, allowing other program blobs to be connected to one of the fan-in

Instruction	Description
`SCG v c`	Sets the cargo bit number `c` to the value `v`.
`JCG c`	Proceeds to bond site "3" if cargo bit number `c` of the ADB is 0, and to "2" otherwise.
`JB b`	If bond site `b` is unbound ($\perp$), proceeds to bond site "3," otherwise to "2."
`INS b1 b2`	Inserts a new blob in the data sequence, connecting the new blob's bond site `b2` to the ADB's bond site `b1`, and `b1` of the new blob to the blob of the ADB's previous `b1`.
`CHD b`	Sets the new ADB to the blob at bond site `b` of the current ADB.
`SWL b1 b2`	Swaps the bond site `b1` of the ADB with `b1` on the ADB's `b2` blob.
`SBS b1 b2`	Swaps bond sites `b1` and `b2` on the ADB.
`SWP1 b1 b2`	Swaps bond sites "1" on the blobs at `b1` and `b2` of the ADB.
`SWP3 b1 b2`	Swaps bond sites "3" on the blobs at `b1` and `b2` of the ADB.
`JN b1 b2`	Sets the APB's bond site `b1` to be the bond site `b1` of the blob at the APB's bond site `b2`.
`DBS b`	Sets the ADB's cargo bits number 0 and 1 to the bond site number on the blob connected to the ADB "in the other end" of bond site `b` on the ADB.
`FIN`	Fan-in instruction. Does nothing but propagate APB.
`EXT`	Halts program.

Table 4.1: *The 13 operation codes in the blob model instruction set. Naming: `v` is a 1-bit value, `c` is a 3-bit cargo index, and `b` is a 2-bit bond site number. All instructions proceed to the next program blob at bond site "2" unless otherwise noted.*

blobs: If i different blobs need a bond to a program blob b, we put $i-1$ fan-in blobs before b and connect the i blobs to them.

The blob instruction set is listed in Table 4.1, along with an informal description of their semantics.

The instruction for creating a new blob, `INS`, works by "grabbing" an unused blob that is not connected to any other blobs. Consequently, a blob computer can be thought of as a large pool of blobs, some of which are organized in a program and a data structure, while others are floating around, acting as "blank" blobs.

We adopt the notation from [54] to textually represent blobs. Let one blob be represented by $B[xxxxxxxx](s_0s_1s_2s_3)$, indicating that its cargo bits are $xxxxxxxx$ and its four bond sites are $s_0s_1s_2s_3$. The default successor bond site is named S, the predecessor P, the bond site between instruction and data *****, and an unused bond site $\perp$.

Example 4.1 The blob $B[11001101](*PS\perp)$ is the program blob for the instruction `SCG` (encoded as 100) with the arguments 1 and 101, meaning that bit position 5 in the ADB shall be set to the value 1. Because the activation bit (the leftmost bit) is set to 1, it is the APB, also evident by its possession of the bond *****. The effect of this operation is illustrated in Figures 4.2a and

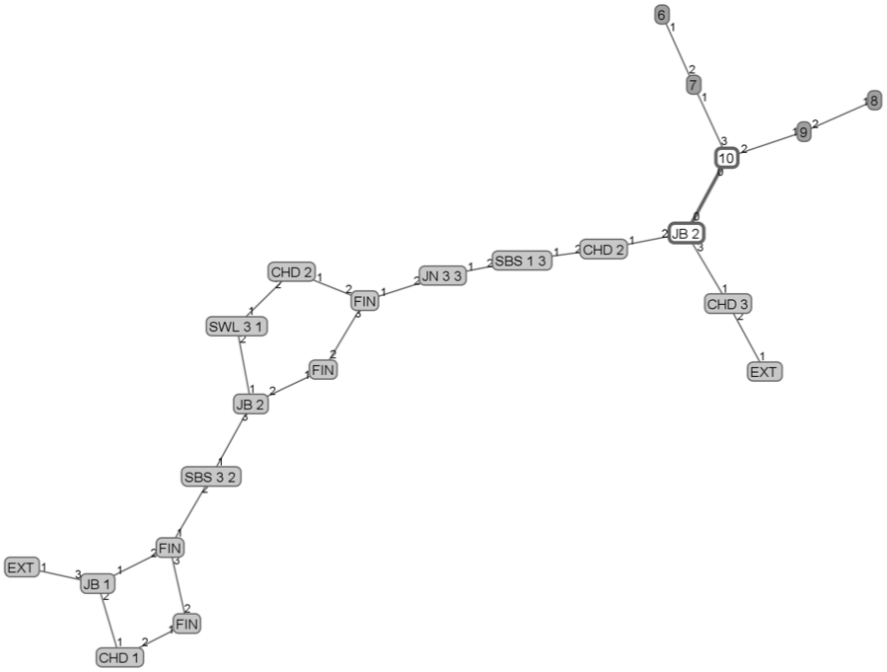


Figure 4.3: *The program "ListAppend," visualized with the tool "BlobVis" [118] (available from http://blobvis.appspot.com/). Program blobs are annotated with their instruction code and arguments, and data blobs are annotated with their cargo bits. The illustration clearly reveals the program's two loops and currently active program and data blob.*

4.2b, representing the program before and after execution of the `SCG` instruction, respectively. As can be seen, the cargo of the ADB is altered from 00000000 to 00000100, but the ADB itself remains the same. `SCG` simply passes the APB-role to its successor blob, evident by the activation bit's change.

Example 4.2 Figure 4.3 shows an example of a "real" program. The illustration shows the program at one moment during its execution, in which the APB is the blob for the `JB`-instruction. The use of the fan-in blobs, `FIN`, is exemplified, as are the creation of basic looping structures. We see that the program has two exit points, represented as two distinct `EXT`-blobs.

4.3 Data Blobs

Data blobs are like program blobs, except that they are not equipped with the same semantics, and that they are treated as binary numbers instead of operations. Only seven of the eight cargo bits are available for data storage, as the first bit is reserved as an activation bit, like on the program blobs. Allowing all eight bits to be used would cause the data and program blobs to be of different types, thus not allowing programs to be treated as data and vice versa: Only one program blob may have its high-order bit set to 1 (by Definition 4.2), and all data blobs with high-order bits set to 1 would violate this if they were

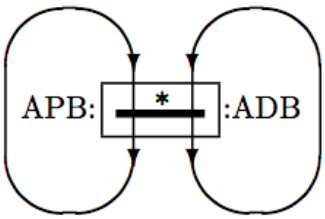


Figure 4.4: *Basic sketch of a blob computer. The left and right oval shapes represent the program and data blob structures, respectively. One program and one data blob are active, the APB and ADB, and are connected at the bond site **. Illustration from [54].*

treated as program blobs. Therefore, the high-order bit is always 0 on data blobs, except in the case where a data blob serves as dual purpose as program blob, and it is the APB.

Bond sites "0" to "3" of the data blobs may be connected to any other data blob (indeed, to program blobs, as program and data blobs are indistinguishable). For the ADB, bond site "0" is always used to connect the currently active data blob to the currently active program blob, as per Definition 4.2.

Both the cargo bits of the data blobs and the structure into which the data blobs are arranged may be altered by the program blobs, as can be seen by studying the `SCG` and various swap instructions in Table 4.1.

4.4 Program Flow

The flow of the program execution emerges from the continued change in position of the bond * between the active data and active program blob. The computation itself is the repeated manipulation of the cargo bits and the bond sites of the data blobs (and potentially the program blobs as well). Because a blob computer consists of nothing but the blob program and the blob data, the distinction between software and hardware does not exist. This relies on the assumption that "something" in the soup of blobs is capable of providing the energy to move the bond sites and alter the cargo bits.

However, the "something" that provides the energy does not perform the actual computations, i.e., it does not move the bond * and it does not alter the cargo bits and bonds of the data blobs. If it did, the blob model would be little more than an esoteric version of a Turing machine, as the blob structure would be akin to a tape on which a foreign device operates. The program blobs perform the computations *by themselves*, using physical capabilities intrinsic to the blobs in the manipulation of data blobs. A useful way of thinking about the required "something" is the presence of heat, allowing a certain set of chemical reactions to take place.

Figure 4.4 is a general outline of a blob computer. Because data and program blobs are the same thing and can be used interchangeably, the structure of the computer may be a lot more intertwined and tangled than that of the illustration.

4.5 Universality of the Blob Model

In [54] an algorithm that can convert any Turing machine description to a blob program, along with an implementation of a self-interpreter in the blob model, is shown. Hence, the blob model is a Turing-universal form of computation because a universal Turing machine can be translated to a blob program, which thus would be able to interpret any other Turing machine.

The algorithm for converting a Turing machine description to a blob program described in Appendix A.3 in [54] uses the basic idea that the tape with n cells is represented as a chain of n data blobs. Each data blob contains in its cargo the symbol s from its corresponding tape cell ($s \in \{0, 1\}$), and the leftmost and rightmost blobs carry a special marker in addition to s. The position of the Turing machine's head is represented as the currently active data blob. Each state transition of the Turing machine may be one of the following two kinds, giving rise to special cases of blob code generation: (i) change the contents of the tape when not in the leftmost or rightmost cell; (ii) change the contents of the tape in either the leftmost or the rightmost cell, resulting in the need for an extension of the tape. The basic compilation strategy is to generate blob code for each state transition and then collect the pieces into one large blob program. For details, see [54].

The fundamental atoms of the procedure are the reading of a Turing machine description and the writing of a blob program's textual representation. Because each primitive action of the Turing machine (move, read, and write) is representable as blob instructions, the combination of them is also representable. Finally, while not explicitly argued in the article it is quietly assumed that this algorithm can be performed by a Turing machine; an assumption that is further justified for the reasons above.

4.5.1 Universality Type

For an analysis of the blob model in the language and context of Chapter 3 to be possible, we must first establish a correspondence between the various Definitions and Concepts, and the notions of the blob model.

- The *computer* is the blobs combined into program and data structures "floating around" in a soup. Concept 3.1 is applied to this notion. In Figure 4.5, the computer is all the blobs illustrated using black circles and the bonds between them.
- An *execution* is, as discussed above, the continuous change in position of the bond *****. This accounts for Concept 3.2. Figure 4.5 depicts an arrow to indicate the movement of the bond.
- One example of a *notion of size* corresponding to Definition 3.4 is the total number of blobs used in the computer, analogous to circuit complexity measures in electronic computers. Using that size notion, the computer drawn in Figure 4.5 has the size 12, as the unused (grey) blobs are not part of the computer.

Definition 4.3 *An* available *data blob is a data blob that is not used in the construction of blob program, i.e., a "blank" data blob. Figure 4.5 depicts available data blobs as small grey circles.*

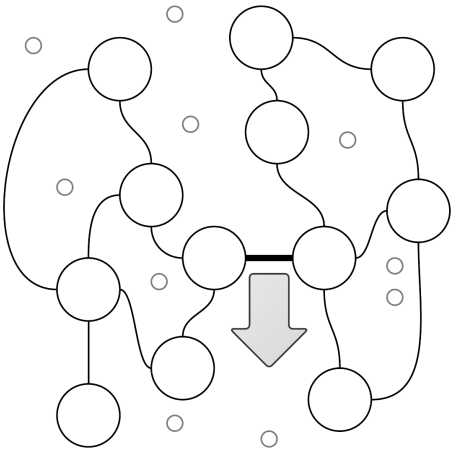


Figure 4.5: *An abstract depiction of a blob computer. The small, grey circles represent "blank" blobs that are not used by the computer. The large arrow is intended to illustrate the flow of execution, namely the change in position of the ADB-APB bond, which is illustrated with the thick line.*

4.5.1.1 Components

A *component*, as specified in Concept 3.3, is one blob. It has as input and output channels the four bond sites, which each has a maximum of one receiver. The input and output communication methods, $\mathfrak{M}_i(blob)$ and $\mathfrak{M}_o(blob)$, are the same, namely the biochemical reactions caused by the blobs themselves. These consist of "reading" or "writing" the contents of a blob and "swapping" bond sites.

4.5.1.2 Configuration

The *configuration* of a blob computer, in the sense of Concept 3.5, is the ordering and contents of a specific subset of the blobs of the computer. Information is carried both in the cargo bits of the blobs and in the way they are connected to each other, i.e., the bond sites. Hence, a configuration is made of the abstract notions "contents" and "connectedness" of some blobs.

4.5.1.3 Assembly Graph

The *assembly graph* of Concept 3.7 refers to the graph structure emerging when blobs are connected through their bond sites. Thus, the nodes of the assembly graph represent single blobs, and the edges represent bonds between the blobs.

The risk of mentally erasing the distinction between what is data and what is program or hardware is high, because the distinction is a purely analytical construction in the blob model, and does not "really" exist. However, it does make sense to maintain some sort of distinction for the blob model, in order to be able to meaningfully reason about the assembly graph. Because the configuration information is stored both in the cargo bits and in the interconnections between certain blobs, it is impossible to alter the configuration without altering the

bonds between some of the blobs. Thus, the blobs whose bond sites need to be changed must be singled out and recognized as being "special."

4.5.1.4 Assembler Function

The *assembler function* ϕ_k from Concept 3.8 is the function that takes a subset of the set of all blobs (there are $2^7 = 128$ different blobs) and a description of a Turing machine T. Using the algorithm above, it connects a number of blobs into a structure that is a blob computer recognizing a subset of the language of T, having only strings shorter than k. Arbitrarily high k can be used, as long as a sufficient number of blobs is available.

To have a useful assembler function, the following notion of automation is introduced:

Concept 4.1 *An* automated method *of connecting together blobs is a method whose steps can be written down and carried out by a person or device acting purely on the basis of the written instructions. Hence, the method produces the same result every time it is run.*

The requirement imposed by Concept 4.1 states basically that no "tricks" can be used when putting the blobs together; the laboratory assistant whom the task has been assigned may not perform any actions as a result of independent thought or opinion.

Definition 4.4 *The following requirements are the* automateable requirements*:*

(i) an automated method of physically connecting blobs together is present;

(ii) the method from (i) is controllable by some Turing machine;

(iii) the algorithm of Appendix A.3 in [54] can be run on some Turing machine.

Theorem 4.1 If the blob model satisfies the automateable requirements in Definition 4.4, then the blob model is *mechanically universal*, and for each $k \in \mathbb{N}$, a blob computer recognizing languages with strings of length smaller than k can be constructed, whence the blob model has *strength* ω.

Proof. Let $n \in \mathbb{N}$, and let Φ_n be as defined in Definition 3.6.

Per the assumptions, the algorithm for translating a Turing machine description to a blob code specification can be run on some Turing machine, and the assembly of blobs can be controlled by some Turing machine. Furthermore, the assembler functions ϕ_k can have arbitrarily large k, provided that enough raw materials (i.e., blobs) are present. Hence, the assembler functions ϕ_k belong to the set of *computable* assembler functions Φ_n. This, together with the fact that it can be done for any $n \in \mathbb{N}$, lets us invoke Definition 3.8 to conclude that the blob model is *mechanically universal* with strength ω. □

4.5.1.5 Well-Behaved Blob Computers

To ensure that a program cannot modify itself, thereby severely limiting the applicability of the cut-idea from Chapter 3, we introduce the notion of *well-behaved* blob computers.

Definition 4.5 *A* simple cycle *is a finite graph $G = (V, E)$ with a cycle containing every vertex in V, such that:*

$$\forall v \in V : \ \deg(v) = 2.$$

Definition 4.6 (Immediate well-behavedness.) *At time t_0 during the execution of the blob computer B, no subgraph of the assembly graph of B exists that is a simple cycle containing both APB_{t_0} and ADB_{t_0}, the currently active program and data blob, respectively.*

Definition 4.7 (Well-behavedness.) *The blob computer B is* well-behaved *if it is immediate well-behaved at all times t during its execution.*

Lemma 4.1 The well-behavedness criterion of Definition 4.7 is equivalent to the following criterion:

- At no point in time may an n-cut $C = (L, R)$, $n \geq 2$, of the assembly graph exist, such that $APB \in L$, $ADB \in R$, and L and R are connected graphs.

Proof. We show that well-behavedness from Lemma 4.1 implicates well-behavedness from Definition 4.7, and that non-well-behavedness from Lemma 4.1 implicates non-well-behavedness from Definition 4.7:

$n = 1$ If only 1-cuts exist, the program is well-behaved by assumption. We cannot choose a subgraph of the assembly graph that contains both APB and ADB while also being a simple cycle: Because APB and ADB are in the subgraphs L and R, respectively, and no other edges from L to R exists, at least one vertex will have an odd degree. Hence, the blob program is well-behaved in the sense of Definition 4.7 for $n = 1$.

$n \geq 2$ Consider the n-cut for $n = 2$. The program is not well-behaved by assumption. It is possible to pick a subgraph containing APB and ADB that contains the two edges in the cut (one of them is the edge between APB and ADB), which is a simple cycle because L and R are connected. For $n > 2$ it is possible to pick the same subgraph as for $n = 2$. Hence, for $n \geq 2$, the blob program is not well-behaved in the sense of Definition 4.7.

□

Consequently, a well-behaved blob program may only modify data in a designated connected subgraph of the blob assembly graph. Figure 4.6 shows two drawings of blob programs, one of which is well-behaved.

Lemma 4.2 The problem of determining whether a blob program is well-behaved is undecidable.

Proof. The blob program on Figure 4.7 is non-well-behaved after the execution of the instruction `CHD 0`. Therefore, a blob program that executes `CHD 0` is non-well-behaved. Additionally, we see from the instruction set that no other instruction is capable of moving the bond between APB and ADB to APB's bond site "0." Hence, a blob program is well-behaved if and only if the instruction `CHD 0` is executed at some point in time, and it is immediate well-behaved at time $t_0 = 0$.

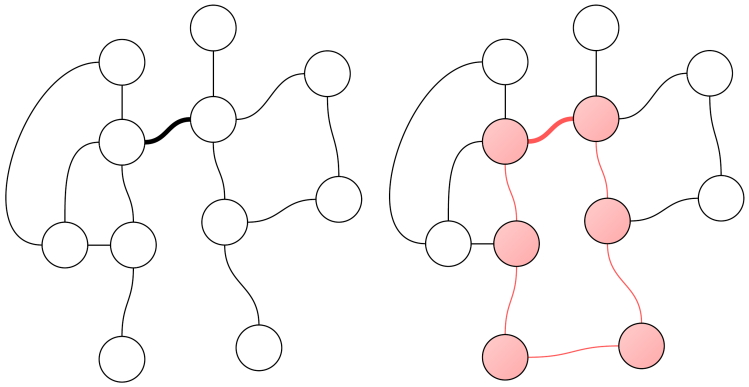


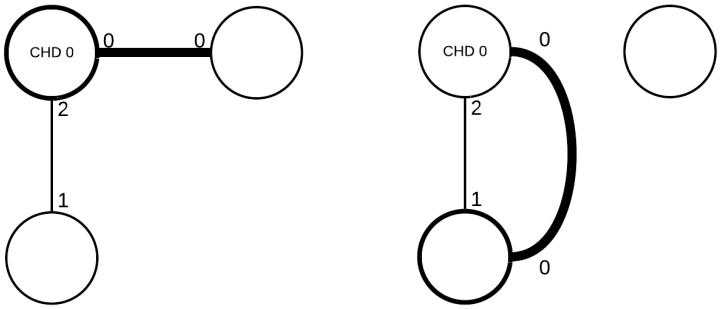
Figure 4.6: *Examples of a well-behaved (left) and an non-well-behaved (right) blob program. The vertices and edges of the simple cycle are highlighted in red, and the vertices representing the APB and ADB are connected with the fat edge. Clearly, a* 1*-cut* $C = (L, R)$ *where* $ADB \in R$*,* $APB \in L$*, and both are connected graphs, exists for the left graph but not for the right one.*

Notice that the Turing machine simulation from [54] does not use the `CHD 0` instruction. Construct a program that alters the textual representation of blob programs by inserting a `CHD 0` instruction in front of each exit point (`EXT` instruction). Assume that well-behavedness is decidable. As per the bi-implication described above, this assumption means that we can decide whether the instruction `CHD 0` gets executed. Thus, we can pick any program P, perform a polynomial-time reduction from P to a blob program that simulates a Turing machine description of P, and decide whether `CHD 0` is executed at any point in time. But since we know that `CHD 0` is *only* present at the exit points of the program, this procedure also decides whether the Turing machine halts, and therefore whether P halts: a contradiction. Hence, well-behavedness is undecidable. □

Though seemingly fatal for practical reasonings about "good" programs, Lemma 4.2 is not a show-stopper. A key observation is that immediate well-behavedness is robust for programs that do not execute `CHD 0`, giving rise to the following Lemma:

Lemma 4.3 If the instruction `CHD 0` is absent from a blob program, well-behavedness is decidable.

Proof. As noted in the proof of Lemma 4.2, a blob program is well-behaved if and only if the instruction `CHD 0` is never executed, and if the program is immediate well-behaved at $t_0 = 0$. Because `CHD 0` is never executed, we can check for well-behavedness by checking for immediate well-behavedness at $t_0 = 0$. This can be done by checking for the existence of a 1-cut in the assembly graph satisfying the criterion in Lemma 4.1, which can be done in polynomial time in the size of the assembly graph on a Turing machine. □



(a) *Before.* **(b)** *After.*

Figure 4.7: *A blob program before and after execution of the instruction* `CHD 0`*. The instruction causes the APB to become the new ADB, thus allowing self-modifying programs. The APB and the bond* ***** *between the ADB and the APB are illustrated with thicker lines.*

4.5.1.6 Programming Function

A self-interpreter constructed using some 50,000 blobs is demonstrated in [54]. Figure 4.8 is a generated visualization of the self-interpreter, and an abstract high-level view of a blob self-interpreter is given in Figure 4.9.

On the need for a self-interpreter. A programming function associates with one particular computer. The function enables the computer A to act like other computers, e.g., the computer B. Hence, the act of programming computer A to do the computations of computer B introduces a layer of abstraction between the actual act of computing and the computer B.

For the discussion about the self-interpreter, we invoke the artificial distinction between the "hardware" and the data mentioned above. One way of doing this is to regard a subgraph of the assembly graph as a component, thereby wrapping the configuration of the computer in one abstract object. Thus, the "data" part of the self-interpreter, i.e., the rightmost two ovals of Figure 4.9, is one giant component, which carries some configuration information, even though it is really just the same thing as the rest, and the configuration information is really just the bonds between the "real" blob components and their cargo bits.

The introduction of this artificial distinction may seem like a foul trick, indicating fundamental problems with the used notions. However, it is not different from, say, the magnetization of some metal, or the punched holes in some cardboard. Our problem is simply that, whereas there is a very clear distinction between the data (holes in cardboard, e.g.) and the hardware (the cardboard itself) in other computing models, this distinction is completely absent in the blob model. The fact that a program is structurally equivalent to a piece of hardware, which is indistinguishable from data, renders the ability to alter data the ability to alter hardware as well!

Furthermore, as an extra layer of precaution, we will assume that the self-interpreter is well-behaved. Taking a look at the structure in Figure 4.9

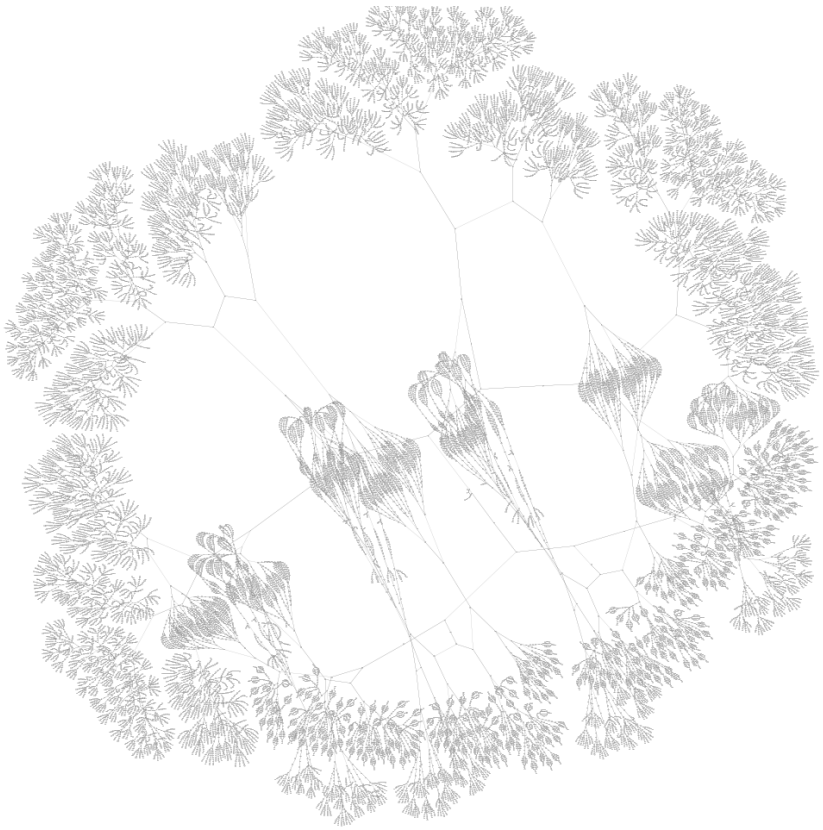


Figure 4.8: *A self-interpreter constructed using blobs. For the sake of readability, the graph is limited to contain only those blobs that are distanced at least 25 bonds from the starting APB. Like Figure 4.3, the graph was generated using "BlobVis" [118].*

seemingly justifies this assumption. Finally, if we disallow the `CHD 0` instruction, the assumption is non-controversial, as it is decidable (Lemma 4.3).

Lemma 4.4 For the blob self-interpreter A_k with $c \cdot k$ available data blobs, where c is some constant, a family of *programming functions* $\psi_0 \dots \psi_k$, corresponding to Concept 3.9, exists.

Proof. The self-interpreter uses as its "tape" a structure of data blobs encoding the instructions and input data of the interpreted program (see Figure 4.9). Hence, the structure and cargo bits of the data blobs represent the configuration of the self-interpreter computer. The amount of data blobs available for this structure is analogous to the number of cells on a Turing machine's tape.

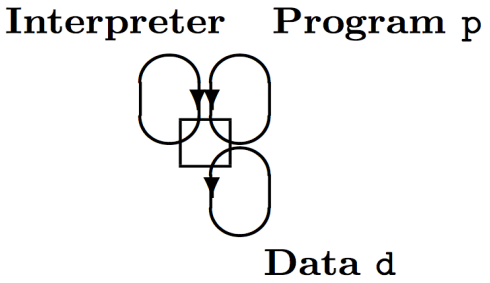


Figure 4.9: *High-level structure of a blob self-interpreter. The interpreter is the leftmost oval, and the two rightmost ovals annotated* `p` *and* `d` *encode the program to be interpreted and its input data, respectively. The programming function alters only* `p` *and* `d`*. Illustration from [54].*

Therefore, for a suitable constant c, the self-interpreter can be programmed by programming functions up to ψ_k when it has $c \cdot k$ available data blobs. □

Theorem 4.2 If the blob model satisfies the automateable requirements from Definition 4.4, *and* there exists $c \cdot k$ available data blobs, for some constant c, then the blob self-interpreter A_k is *linguistically universal* with strength k.

Proof. Let $k \in \mathbb{N}$, and let Ψ_k be as defined in Definition 3.7. Per Lemma 4.4 we have that programming functions $\psi_0 \dots \psi_k$ exists.

By assumption (i) and (ii) of Definition 4.4, a Turing machine that can modify blobs exists. Furthermore, the requirements for Theorem 4.1 is satisfied by assumptions (i)–(iii) of Definition 4.4, meaning that we can convert any Turing machine description into a blob program. Consequently, the programming functions $\psi_0 \dots \psi_k$ are members of Ψ_k, whence the self-interpreter A_k is *linguistically universal* with strength k by Definition 3.9. □

Theorem 4.3 If the blob self-interpreter A_k is well-behaved, it has *localized control.*

Proof. There exists a cut $C = (L_{A_k}, T_{A_k})$ of the assembly graph of the self-interpreter A_k such that (i) L_{A_k} represents the components that make up the control mechanism of A_k (the leftmost oval shape on Figure 4.9), (ii) T_{A_k} represents the components making up the tape of A_k (the two rightmost ovals on Figure 4.9), and (iii) the underlying structure of T_{A_k} can be altered and increased in size to T_{A_n}, $n > k$, without altering the underlying structure of L_{A_k}. Hence, per Definition 3.11, the blob self-interpreter has *localized control.* □

Theorems 4.2 and 4.3 taken together show that the self-interpreter is linguistically universal with arbitrarily high strength, as its memory can be extended easily.

Technique	Assembler function	Programming function	Mechanically universal	Linguistically universal	Localized control	Generates all solutions	Implemented
Blob Model	Yes	Yes	Yes	Yes	Yes	No	No
Adleman–Lipton Method	No	No	No	No	No	Yes	Yes
Hairpin Formation	No	No	No	No	No	Yes	Yes
Boolean Circuits	Yes	?	Yes	?	?	No	Yes
Self-Assembly	Yes	?	Yes	?	?	No	Yes
E. Coli Recombination	No	No	No	No	No	Yes	Yes
Whiplash PCR	No	No	No	No	No	Yes	Yes
FokI Restriction	Yes	?	Yes	?	No	No	Yes

Table 4.2: *Comparison of different biological models for computation. There is a correlation between* not *generating the entire solution space and* maybe *having a programming function.*

4.5.1.7 Assembling vs. Programming Blob Computers

As a consequence of the equivalence between software and hardware, the blob model has the peculiar feature that it is more efficient and simple to use it as a mechanically universal computer, that is, to invoke the assembler function for each new problem, than to use it as a linguistically universal computer. The difference in efficiency originates from the programming function's need for a self-interpreter, which causes a fixed overhead per instruction: The real APB is bonded with the real ADB, which represents the simulated APB that is again bonded with the simulated ADB. Thus, using the programming function as compared to using the assembler function is analogous to running a piece of code on a virtualized PC versus running it directly on a real PC.

4.6 Comparison to Other Biomolecular Methods

After having established the correspondence between the notions of Chapter 3 and the blob model, we compare the model to other biomolecular computing techniques. Chapter 2 offers more detailed descriptions of the techniques, and Table 4.2 gives an overview of the differences between the techniques discussed here.

The set of methods selected for comparison here is necessarily a proper subset of all the implemented methods from the literature. However, based on the earlier literature survey, it is our impression that this subset provides an overall impression of the relation between the blob model and previous methods, most notably the exceptionality of the blob model's programmability.

The following discussion uses the property "Generates all solutions." Due to space efficiency, this is shorthand for "Generates all solution candidates," i.e., also candidates which turn out *not* to be solutions.

4.6.1 Adleman–Lipton Method

A computer constructed using the Adleman–Lipton method has as its components the individual DNA strands that represent a part of a solution. Hence, for the calculation of a Hamiltonian path, the components are the DNA strands representing the edges of the graph.

Assembler function. The general idea behind the Adleman–Lipton method is to generate the entire solution space and then use some filtering technique to remove the incorrect solution candidates. The particular implementation of this strategy in the Adleman–Lipton method suffers from the problem that is does not scale: As noted in [53], the amount of DNA required for a 200-vertex graph has a weight greater than that of the Earth. Furthermore, for more general problems, i.e., any Turing machine, it is not obvious how to simulate it, even when considerations about physical constraints are ignored. One could imagine an approach in which the DNA strands represent Turing machine tape configurations, and the filtering technique ensures that only halting configurations remain.

Based on the problems with scalability and Turing machine simulation, we conclude that this technique has no general assembler function.

Programming function. A computer constructed with the Adleman–Lipton method cannot have a programming function. A programming function rearranges the configuration of some computer, leaving the computer itself intact. But there is no configuration to alter; the computational model works by changing the computer itself, by filtering away incorrect solutions.

Mechanically universal. Because there is no assembler function, the technique cannot be mechanically universal.

Linguistically universal. Likewise, as there is no programming function, the technique cannot be linguistically universal.

Localized control. The memory of a computer constructed with this technique is distributed throughout all its components. There is no single place where information is stored, rather, it is the sum of the remaining DNA strands that define the contents of the memory. Hence, the computer has distributed control.

Generates all solutions. As mentioned above, the method generates all solution candidates.

4.6.2 Hairpin Formation

Hairpin formation was used to solve 3-SAT problems by letting wrong solutions fold hairpin structures that are recognizable by an enzyme. The components of a computer employing hairpin formation are the solution candidate strands, that is, each set of truth value assignments.

Assembler function. Hairpin formation utilizes the same basic strategy as the Adleman–Lipton method, namely generating the solution space and

applying some method to eliminate incorrect solutions. Hence, the scalability issues are likely to reappear here, causing an unrealistic amount of DNA to be required for any "real" problems. Likewise, how to do the simulation of any Turing machine is not clear. Hence, we note that the method has got no assembler function.

Programming function. Like with the Adleman–Lipton method, a computer using hairpin formation as its working principle has no means to alter its configuration, because the computation proceeds by physically reducing the computer, ending with a concentration of correct answers that can be detected. Hence, there is no programming function.

Mechanically universal. Owing to the fact that no assembler function exists, hairpin formation is not mechanically universal.

Linguistically universal. No programming function exists, whence hairpin formation is not linguistically universal.

Localized control. Increasing the problem size forces one to increase the size of all the candidate solution strands, that is, every component in the computer, thus demonstrating that hairpin formation has distributed control.

Generates all solutions. Hairpin formation generates all solutions, as discussed above.

4.6.3 Boolean Circuits

Biological Boolean circuits have been implemented and tested in various different ways. Common for all of them is that basic logic gates are constructed, emitting special signaling molecules, e.g., DNA marker strands, for other gates to operate upon. The size of a circuit is therefore limited by the number of different signals it can produce.

Assembler function. Assuming that it is possible to propagate signals to arbitrarily many biological Boolean gates, an assembler function for biological Boolean circuits exists. This conclusion is reached from the fact that any Turing machine can be represented as a Boolean circuit, and under the mentioned assumption we are able to construct any biological Boolean circuit.

Programming function. On the other hand, it is not clear whether a programming function exists. There is a possibility that a programmable circuit can be constructed, with an ability to accept some form of input and moderate its behavior based upon that. This would correspond to implementing a CPU with biological Boolean gates.

Mechanically universal. As there is an assembler function under the assumption that arbitrarily many different signals can be sent, biological Boolean circuits must be, like ordinary Boolean circuits, mechanically universal.

Linguistically universal. If a programmable circuit like described above could be constructed, we would have a linguistically universal computer with finite strength k.

Localized control. A circuit that is programmable might also have localized control. As it is conceivable that the circuit also has a part that works as a memory unit, the size of this memory unit could be adjustable, and the computer would have localized control.

Generates all solutions. This method does not generate all possible solution candidates.

4.6.4 Self-Assembly

In a computer that uses self-assembly of DNA tiles, the components of the computer are the tiles themselves.

Assembler function. Because any Turing machine can be simulated by Wang tiles, DNA tiles emulating Wang tiles can simulate any Turing machine by self-assembly. Consequently, an assembler function for DNA tiles exists, namely the generation of a set of tiles that self-assemble in such a way that they represent a halting configuration of the Turing machine.

Programming function. It is not clear whether a cellular automaton with a programming function exists. The intrinsically local actions performed in the blob model seem to suggest that something similar could be done with a cellular automaton. If that is possible, it would not be inconceivable that it can be done with a DNA tile self-assembly version of the automaton.

Mechanically universal. As there is an assembler function that allows the simulation of any Turing machine, the self-assembly of DNA tiles is mechanically universal.

Linguistically universal. If a set of DNA tiles with a programming function could be constructed, a linguistically universal DNA tile set with strength k could be constructed.

Localized control. Moreover, localized control is not inconceivable, given an appropriately constructed set of DNA tiles, allowing for a separation of "memory tiles" and "work tiles."

Generates all solutions. Self-assembly with DNA tiles does not generate every candidate solution.

4.6.5 E. Coli Recombination — "Bactoputing"

Successful demonstrations of *E. coli*-based DNA recombination have been constructed, as mentioned in Chapter 2, based on a "miniature" version of the Adleman–Lipton method that occurs in the interior of each cell.

Assembler function. The fundamental computational mechanism relies on the approach wherein every possible solution candidate is generated and then scrutinized for correctness. The by now well-known problems regarding scalability and simulations of general Turing machines therefore arise in this context as well. Again, the approach is unlikely to allow for the simulation of general Turing machines, leading us to the conclusion that no assembler function exists.

Programming function. Due to the fact that the fundamentally same approach as in the Adleman–Lipton method is used, a programming function does not exist. Like previously, the problem is that the computer consumes itself in the process of computing, thus not allowing us to alter a configuration and let the computer operate based on that.

Mechanically universal. No assembler function exists, and hence the technique is not mechanically universal.

Linguistically universal. On a similar note, no programming function exists, whence bactoputing as understood here is not linguistically universal.

Localized control. The memory belonging to a computation resides in all the solution candidate strands. As a result, increasing the size of the problem instance requires us to increase the size of every component, which is contrary to the requirements of localized control. Therefore, the method has distributed control.

Generates all solutions. Bactoputing (as understood here) generates all possible solution candidates.

4.6.6 Whiplash PCR

A whiplash PCR-based computer consists of a library of all candidate solutions, encoded as DFAs. Those DFAs that can reach their final states represent correct solutions. Consequently, the components of the computer are the DFAs and the enzymes catalyzing the whiplash PCR reaction.

Assembler function. Designing a whiplash PCR-based computer relies on the generation of the entire solution space, and searching it by exploring which ones allow for transitions to the accepting state. Simulating a Turing machine using this method would require the machine's memory to be represented in the state set, as well as the generation of every possible tape configuration for the machine. As the possible number of tape configurations of a Turing machine is infinite, this cannot be done. Thus, we conclude that whiplash PCR does not have any assembler function.

Programming function. No configuration that can be altered exists, because the computer is the sum of all the parts that each represents an attempt at a solution, combined with the machinery that filters away the incorrect ones. Therefore, when changing the problem that the computer works on, one changes the computer itself. Consequently, no programming function exists.

Mechanically universal. Because there is no assembler function, whiplash PCR-based computers are not mechanically universal.

Linguistically universal. Likewise, due to the lack of a programming function, whiplash PCR-based computers are not linguistically universal.

Localized control. A computer constructed using this approach does not possess localized control; the "control mechanism" and the "tape" are both present in the DFAs, as they encode their own transition table as well as their current position.

Generates all solutions. The approach generates all possible solution candidates prior to filtrating away the wrong ones.

4.6.7 FokI Restriction

A *FokI* restriction-based computer uses as its components the transition molecules that repeatedly chop off parts of the input DNA strands, and the *FokI* restriction enzyme itself. The method could in principle also be used with other types of restriction enzymes, as long as they cut the DNA strands asymmetrically, leaving sticky ends.

Assembler function. An assembler function for *FokI*-based computers exists, contrary to the previously discussed DFA method. This is due to the fact that the method does not rely on the generation of all possible solution candidates, rather, a set of transition molecules is created and the input strings are mixed with them. In conclusion, the assembler function constructs the set of transition molecules required to recognize the desired input strings, represented as DNA strands.

Programming function. The method's authors use the term "programmable" in a different sense than what is meant in this context. Their notion of programming is to construct an entirely new DFA capable of performing one computation, or state change sequence, which corresponds to what we would here call the assembler function.

It is not clear whether a programming function is possible on a *FokI*-based computer. If it was possible, however, it would be rather restricted: Because the *FokI*-computer is a DFA, we would have to encode a universal Turing machine as a DFA, at the expense of an explosion of the state count. This is impossible for a truly universal Turing machine, but possible for a fixed-size approximation. That Turing machine would then only be able to simulate Turing machines with encoding lengths up to some constant n.

Mechanically universal. The technique is mechanically universal, as any state transition table of a Turing machine can be encoded as transition molecules in a *FokI*-computer, and a finite subset of the Turing machine tape can be represented as an increase in the number of states.

Linguistically universal. Given a programming function, as discussed above, a *FokI*-based computer might be linguistically universal with strength n.

Localized control. The technique has distributed control, as the only way to increase the amount of memory of an automaton is to alter all states, that is, all the components with the exception of the restriction and ligase enzymes.

Generates all solutions. A *FokI*-based computer does not generate the entire solution space.

4.7 Degrees of Automation

Some of the computing techniques mentioned here have automated methods of assembly. This does not mean that every implementation of the techniques actually used an automaton which carried out the work of assembling the components of the computer, rather, it is possible to build such a device.

Some of the authors of the techniques mentioned above use the quantity "laboratory steps" to describe the difficulty or complexity of setting up or executing a computation. The notion of a laboratory step is intuitively defined as the basic actions that a laboratory assistant needs to do in order to perform the method.

Even though the quantity "laboratory step" is not exactly defined, it does provide some impression of the complexity of a method. When building a machine capable of assembling biological components, it must also perform the laboratory steps that were previously performed by humans, for instance as a "lab-on-a-chip" implementation. Hence, it does make sense to consider laboratory steps, but the focus must be wider than only on the somewhat vague notion of a "step," as other factors are relevant too. We attempt to consider some of these "steps" from three different perspectives: The amount of material required, the amount of time required, and the scalability of the steps.

- Adleman notes in his 1994 paper that the number of laboratory steps grows linearly in the problem size [3]. However, a laboratory step can in this case be the performing of PCR, a process which takes time measurable in minutes or hours. The entire solution space is generated, which required an impractical amount of material for larger problem instances, as previously mentioned.

- Sakamoto et al. note that several transition steps of a whiplash PCR-based DFA can be considered as one laboratory step, thereby making the required amount of laboratory steps independent of the problem instance size [99]. However, like the Adleman–Lipton method, this method relies on the generation of the entire solution candidate space. This material requirement turns problematic for larger problem instances, both in the sense that too much DNA is required, but also on the time scale: generating solution candidates takes time.

- Qian and Winfree report that for their implementation of a circuit computing the square root of a four-bit number [88] they required: Approximately one week of preparation time, wherein different DNA complexes of the construction were annealed and purified; one hour to assemble the circuit; five minutes to input data (mixing input strands in the test tube); and finally up to 10 hours waiting until the results could be read out using fluorescent molecules. In total, it took about eight days to compute the square root using that method.

4.8 Main Problem

We conclude the discussion of the blob model with an explicit formulation of the main problem related to the blob model, viz., that of its implementability:

- *Which implementation substrate is best suited to implement the blob model?*
  - *Which implementation substrates can implement a programming function?*
  - *Which implementation substrates can construct a linguistically universal computer?*

From the preceding discussion and Table 4.2 we get the impression that, of the techniques discussed here, *biological Boolean circuits*, *DNA self-assembly*, and FokI *restriction* stand the best chance of becoming an implementation substrate for the blob model, as these techniques may be able to possess the characteristics that are special for the blob model, which the Adleman–Lipton method, for instance, cannot.

5 Implementation Substrates

In this chapter we present the implementation substrates from the previous chapter as a basis for an implementation of the blob model, and discuss the feasibility of such an implementation. The treatment considers the approaches that Table 4.2 indicates are possible implementation substrates: DNA self-assembly, biological Boolean circuits, and *FokI* restriction.

5.1 Requirements Imposed by the Blob Model

In order to be able to implement the blob model, there are some basic "core" operations that an implementation substrate must be able to do. The viability or realizability of the blob model given a specific substrate will be discussed based on the dimensions listed below.

5.1.1 Instruction Laboriousness Hierarchy

When attempting to assess whether a given substrate or technique is usable for implementation purposes or not, it is convenient to have some sort of "laboriousness hierarchy" of the blob instructions. Thus, if we can build the "difficult," or more laborious, operations, we should also be able to build the "easy" ones. Due to the experimental nature of the blob model, such a hierarchy is necessarily informally described. The ordering of the instructions in a hierarchy is new; the original article does not operate with such a distinction.

We classify the blob model instructions into six distinct classes of laboriousness. In the order of easy to more difficult:

1. The instruction `EXT` is the easiest instruction, as it does not perform any actions.
2. The next level contains the instruction `FIN`. As it only serves a structural purpose, allowing multiple fan-ins to blobs, it is deemed "easy."
3. The instruction `CHD`.
4. Instruction `SCG`.
5. Instructions `SWL`, `SBS`, `SWP1`, `SWP3`, and `JN`.
6. The jump instructions `JCG` and `JB` along with the instructions `DBS` and `INS`.

We see that the easiest instruction which also performs some interesting task (that is, something else than propagating control flow or halting the program) is the `SCG` instruction. This instruction is therefore used as a "litmus test" of the viability of an implementation substrate in the forthcoming analysis.

5.1.2 Requirements

Figuring out whether an implementation substrate is realistic must involve discussions of how to do the following:

Move the APB-ADB bond It must be possible to move the bond between the ADB and APB, i.e., change which implementation substrate components represent the APD and ADB.

Cargo storage. The substrate representing a blob must be able to store the cargo bits described in the blob model.

Instructions. Naturally, every instruction in the blob instruction set should be supported by an implementation, but the demonstration of how to implement just one of them is a necessary first step. After the problems outlined above have been addressed, the next step is to show how to perform one of the basic instructions. As discussed above, we pick the `SCG` instruction, specifically `SCG 1 5`.

Finally, the considerations outlined in Section 3.3 on page 38 also apply for the blob model, although some of them are implied by the considerations outlined here: Memory I/O is implied by the cargo bits, signal propagation is implied by the movement of bonds, and extendability is present given the complete blob instruction set due to the existence of a self-interpreter capable of "grabbing" new blobs. Moreover, some of the considerations on page 38 are only meaningfully solvable given solutions to the "core" requirements of the blob model enumerated here.

5.2 DNA Self-Assembly

At a first glance, DNA tiles and blobs seem related. Both have four connections with other elements: sticky ends on the DNA tiles and bond sites on the blobs.

However, techniques relying on properties intrinsic to the DNA structure are poorly matched to the blob model because of the inherently distributed, probabilistic nature of the computation steps utilized in DNA based models. Specifically, the dependance on hybridization and denaturation processes are problematic from the blob perspective.

5.2.1 Local Hybridization and Denaturation

Mapping a blob to a DNA tile would require a fine-grained control over the breaking and forming of hydrogen bonds between DNA strands. Specifically, if the Watson–Crick pairing between, e.g., two 20-mers represent a bonding site between two blobs, we need to ensure that only the bonding sites relevant for the currently active program and data blob are changed during program flow. As the forming and breaking of this type of bond are accomplished through cooling and heating the DNA solution, respectively, this turns out to be difficult. The problem is that we are breaking all bonds if we simply heat the solution.

Figure 5.1 is an attempt to provide an impression of just how small an area that must be heated, and whose surroundings must be unaffected (meaning that they will not be affected more than a certain threshold). Based on the

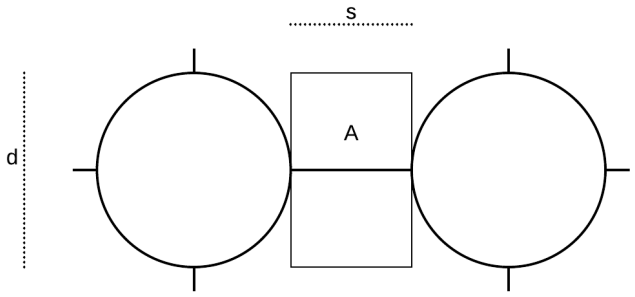


Figure 5.1: *Let d be the diameter of a blob and s the length of a bond. The area A with the size $s \cdot d$ must be treated such that the bond inside the rectangle is broken or created by denaturation or hybridization, respectively, and everything outside it remains unaffected. In [129], the reported sizes are roughly $d \approx 15$ nm and $s \approx 5$ nm. That gives an area with the size $A \approx 75$ nm^2. Note that this is under the assumption that the blobs are laid out in a two-dimensional layout. If we instead consider the three-dimensional box between the two blobs containing just the bond between them, we get a volume of roughly $d^2 \cdot s = 1125$ nm^3.*

dimensions reported in [129], we assume that a blob has the diameter 15 nm and that the length of a bond is 5 nm, necessitating an area of about 75 nm^2 to be treated. This is assuming that the blobs are organized in a two-dimensional structure. If that is not the case, we have to take into account all three spatial dimensions, and we consider a volume of 1125 nm^3 instead.

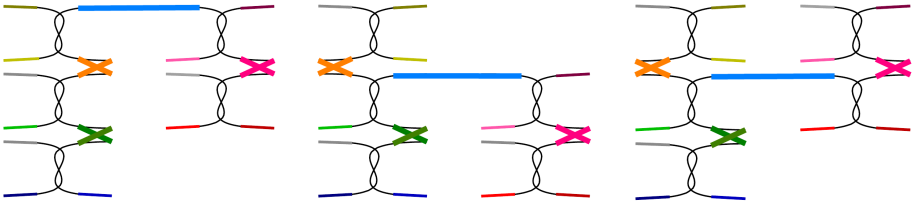
Concept 5.1 (Local thermal change.) *By a* local thermal change *of a volume V we mean the following: V contains two spheres, S and S', such that S' is entirely contained in S, and everything inside sphere S' reaches a specified temperature, while everything outside sphere S is unaffected.*

The use of the sphere in the above definition is due to its being simple to imagine; in any real-world application the "sphere" can be any other topologically equivalent shape.

As we see, the size approximations of Figure 5.1 depict local thermal where the size of the outer sphere is about 1125 nm^3, and the size of the inner sphere is smaller but just big enough to engulf the entire bond structure. Of course, the thermal change required in the scenario of Figure 5.1 corresponds to that required for the breaking or mending of one bond: Some instructions will require local thermal changes of sizes larger than this, as more than one bond must be broken and mended. Still, this just represents multiple applications of the depicted thermal change.

This is probably the greatest obstacle posed by the DNA tile model. Relying on the repeated heating and cooling seems unlikely to provide a passable way to tile-based blob computers, because of the "bulk" approach inherent in the method. Furthermore, an introduction of some kind of "local heating" capability is improbable, as the heat will dissipate to the surroundings that contain other bonds due to the small size.

One way this could be resolved, though, would be to adopt some kind of "color coding," coloring every bond between tiles/blobs uniquely by choosing



(a) $t = 0$ **(b)** $t = 1$ **(c)** $t = 2$

Figure 5.2: *Illustration of the development of the execution of a simple blob program implemented with DNA tiles, utilizing colorings to control the flow. All bonds are marked with a crossing of the color codes, and the long light blue bond is the APB-ADB bond. At time $t = 1$, the APB has changed, but the ADB is the same. The ADB has changed at $t = 2$, where the APB is unchanged.*

different sticky end sequences. This would not change the fact that every single bond would be broken and hybridized repeatedly, but it would (in theory at least) cause the bonds to always be hybridized with the correct partner after a heating process, that is, a bond that is not the APB-ADB bond remains the same after a heating or a cooling process. An obvious limitation of this solution is that a unique sticky end is needed for each blob, of which there can be many (50.000 in the self-interpreter, for instance). Furthermore, the test-tube environment in which the computer is likely to live is noisy, opening up the risk for partly hybridized sticky ends between "unmatched" tiles.

Another way of "closing" the sticky end representing the site for the APB-ADB bond for all tiles except the two correct ones could be to let the sticky end representing bond site "0" be folded up to a hairpin structure. That way, all bond site "0" representations that do not belong to the tile for the active program blob would hybridize with themselves, thereby not forming any "wrong" APB-ADB bonds. This, however, raises the question of how to unfold the sticky end that is to take part in the next APB-ADB formation, something which raises the original question of localized denaturation and hybridization once again. Additionally, if we rely on denaturation we have yet to address the issue with the bond sites "1"–"3"; even though bond site "0" is a hairpin structure, and therefore presumably safe from unwanted pairings, we must still ensure that the rest of the bonds stay in place.

Apparently, the solutions sketched here still rely crucially on the ability to have full, fine-grained control over the forming and breaking of the hydrogen bonds between DNA strands.

5.2.2 Moving the APB-ADB Bond

In order to support moving the APB-ADB bond, the model has to be extended to allow the sticky ends to "change color," that is, change the sequence of nucleotides. Figure 5.2 illustrates the principle of color changing DNA tiles. It is not clear how to do this, as problems regarding localization arises.

One possible method could be to use some approach based on restriction enzymes, cutting the sticky ends in a specific way. This is a similar idea as

Figure 5.3: *Two DNA strands that are connected through their matching sticky ends, represented as the green color. The cargo bits are carried in the blue and red colors. The color schemes for the sticky ends and the cargo bit part must be disjunct, in order to avoid overlaps that cause undesired connections.*

in the *FokI*-based automaton. Still, we must make sure that only the correct strands are being cut. That is, the recognition site for the restriction enzyme must be present only on the strands connecting the APB and ADB, and it must be moved according to the program flow. Furthermore, relying on the repeated shortening of the sticky ends is undesirable, as some blobs may be activated, viz., become the APB, several times during an execution, and because data blobs need to accessible multiple times, for instance when representing a counter value.

5.2.3 Cargo Storage

A blob's behavior is determined by its cargo bit value: The eight bits can encode the instructions `SCG`, `CHD`, etc., each of which gives rise to different types of behavior of the blob. The cargo bit value can also be interpreted simply as an integer in the range 0–255, under which construal the blob's behavior is constant: a data blob containing the integer 55 behaves no differently from a data blob containing the integer 200.

Consequently, we need a way to store the blob's cargo bits that sometimes alters the behavior and other times does not. To this end, attaching a "module" to a DNA tile does not suffice, as the behavior of the DNA tile is determined by the colors on the four sides of it, which would not be affected by the attachment of a module carrying eight bits.

Storing the cargo bits directly in the coloring seemingly allows the desired coupling between the behavior of blobs and the behavior of DNA tiles. As there exists 256 different cargo values, we require 256 different colors for the sides; the different colors elicit different actions. Still, we must also address the scenario wherein a DNA tile contains raw data, i.e., when the cargo bits are interpreted as a number and not as an instruction. In that case, different kinds of tiles may interact with it, forcing us to provide some common "connector" between the tiles. Consequently, the sticky ends contain a part that is an encoding of an eight-bit integer and another part that works as a more general sticky end, allowing different kinds of instruction blobs to attach to data blobs with different payloads. Figure 5.3 is an attempt to illustrate this notion of divided sticky ends.

5.2.4 Implementing `SCG 1 5`

Recall that the instruction `SCG 1 5` sets the cargo bit number five of the ADB to the value 1, and passes the control to the program blob at bond site "2" of the APB, while letting the ADB stay the same. We enumerate a high-level

view of some of the subtasks present in the execution of this instruction in a hypothetical tile-based blob computer implementation:

1. Alter cargo bit number five of the ADB. If we adopt the idea that cargo bits are encoded in specific DNA strands, as briefly discussed above, we require a method to change the ordering of base pairs in substrands of DNA.

2. Alter the cargo of the APB to reflect the fact that it is not active anymore. This requires changing the high-order bit from 1 to 0, and it imposes similar problems as the changing of the cargo value of the ADB. However, this could be remedied by limiting the cargo to be seven bits instead of eight, and representing the active/inactive state of the APB in some other way.

3. Release the APB-ADB bond. As discussed previously, this poses a serious problem, due to the non-local nature of the available techniques for breaking the bonds between tiles.

4. Form a new bond between APB' (the blob at the end of bond site "2" of the APB) and ADB. Again, this requires the ability to locally hybridize DNA.

Hence, if the following three requirements were met:

- a method for localized hybridization and denaturation, capable of local thermal change of volumes with sizes comparable to 1125 nm^3;
- a method that permutes nucleotides in some controllable way, enabling us to "change colors" on the sticky ends;
- some way to control the execution of the two techniques above through the cargo bit configurations;

we would be able to implement at least the `SCG 1 5` instruction and the `CHD`, `FIN`, and `EXT` instructions due to the hierarchy introduced earlier.

5.2.5 Effectiveness

If we managed to solve the problem of how to do local hybridizations, we would face another one: performance issues. Given that hybridization and denaturation occurs on the gamut between approximately 40°C to 90°C, and the change in temperature must be "slow" [4, Chapter 10, p. 332], an amount of time unacceptable for any practical use would be required even for smaller programs, as denaturation and hybridization would be repeated for each performed instruction. This is a general problem that applies to all techniques relying on the massive parallelism inherent in the "melting" of DNA. Consequently, any solution to the problem of local hybridization must also rely on some other method than just heating and cooling to break and mend the bondings between DNA strands.

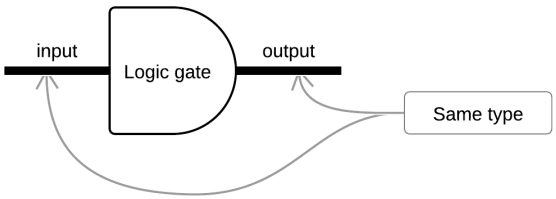


Figure 5.4: *A gate must be able to take as input the output of other gates if large-scale circuits are to be built.*

5.2.6 Feasibility

As should be evident from the the above discussions, the DNA tiles approach poses several challenging problems as an implementation substrate for the blob model, causing it to be unlikely as a realistic candidate for implementation. Specifically, the list of requirements formulated above seems unlikely to be implementable, due to their localized nature.

5.3 Biological Boolean Circuits

In evaluating biological Boolean circuits as an implementation substrate for the blob model, we choose those variants of circuits from the literature that have the following properties:

1. a circuit should compute "unmanaged," not requiring any interaction with humans throughout its computation process, thus fulfilling the requirements of an automaton from Concept 1.1;

2. the gates are "composable," such that gates can be combined to represent increasingly more complicated functions — illustrated with Figure 5.4;

3. the technique represents an attempt at designing a computer, thus having focus on the computational aspects instead of the biomolecular aspects of its uses;

4. an implementation has been demonstrated.

Based on these specifications, we choose: seesaw gates [88, 87], the gates presented by Seelig et al. [100], deoxyribozyme-based gates [106], and enzyme-based gates [11, 78]. In the case with deoxyribozyme-based gates, specification two is ignored.

Although other implementations of Boolean gates have been presented, we have chosen to focus only on those presented here, as the other methods have not been in accordance with the requirements formulated above, summarized by Table 5.1.

The gates are evaluated in the context of an assumed implementation of the blob model in Boolean circuits, assessing the feasibility of using them as components in the hypothetical implementation.

Gate type	Conflicts with
DNA-based [81, 82, 5]	Requirement one, due to the required interaction with a laboratory assistant.
Surface-based [109]	Requirement two: Two circuits cannot be combined easily.
RNA-based [62, 35, 110, 94, 126]	Requirement three: Focus is on gene expression in *in vivo* cells, the natures of the methods "happen" to mimic Boolean logic as a by-product.
Protein-based [114]	Requirement four; no report of an implementation is given.

Table 5.1: *The gate types that are left out of the treatment, owing to their conflicting with the requirements that have been specified.*

5.3.1 Seesaw Gates

The seesaw gates by Qian and Winfree rely on the hybridization of "wire"-DNA with special DNA strands, designed such that they can implement logic gates when combined. Each gate is represented as a concentration of DNA strands with specific sticky ends, where each strand that represents a particular gate has the same "identifier" sticky end. Connections between gates are modeled by the potential signals attached to the gate strands, such that gate A upon activation releases a DNA strand that matches the identifier sticky end for gate B, giving rise to a chain reaction. Thus, modeling connections is a question of letting gates release signal strands that interact with the gates that they should be connected to. In Figure 5.5 this can be seen in part B, where gate "S2" connects to gate "S5." Thereby the reaction is controlled, and the different levels of the circuit are activated in the correct order.

The design of the seesaw gates results in a use-once restriction on the gates: When a signal is emitted from a gate, it is "used" and hence unable to repeat the process. Another consequence of the design is that gates are reversible: Signaling strands between gates A and B can detach from gate A to attach to gate B only to detach from B and reattach to A.

The fact that a circuit destroys itself upon use severely limits the seesaw gates as an implementation substrate candidate. Even when given a hypothetical mapping between seesaw gates and blobs, in which a blob is implemented by a certain group of gates, and each gate have a unique marker, we are forced to let the blobs be use-once as well, thereby eliminating any possibility for, e.g., looping structures. Consequently, such an implementation of the blob model would not be as computationally strong as the "real," theoretical blob model.

Another problem with the seesaw gates is that each gate requires a significant amount of time to react and produce an output. Part C of Figure 5.5 shows that an OR and an AND gate require three and six hours, respectively, before the output is produced, assuming that "on" is defined as between 0.8 and 1.0 (which it is in [88]). The authors remark that the reaction times grow linearly in the depth of the circuit. As our hypothetical blob implementation would most likely consist of several gates for each blob, the time required to execute one instruction would be impractical: Minimum six hours, under the assumption that only a single-layered circuit with at least one AND gate is used.

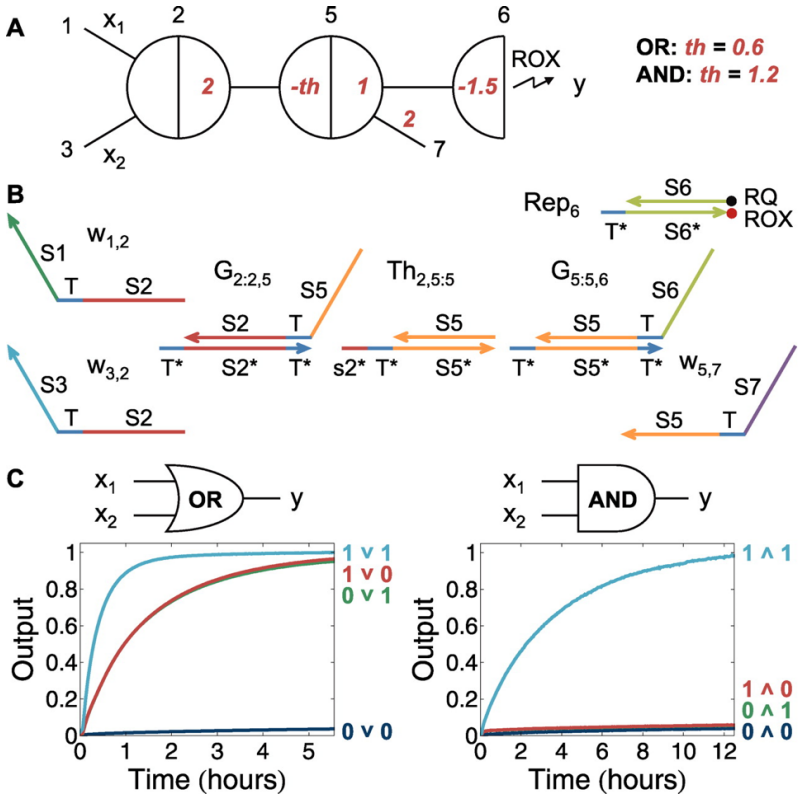


Figure 5.5: *Seesaw gate implementation of either an AND or an OR gate, depending on the threshold values chosen (denoted by* th *in part A). Part A shows a schematic view of the gate. Inputs x_1 and x_2 enter gate 2, if they surpass the threshold* th *gate 5 will propagate signals releasing the fluorophore* ROX*, causing a molecule to fluoresce. Part B is a diagrammatical overview of the DNA implementation where shared colors indicate Watson–Crick complementarity. The connections shown in part A are represented by Watson–Crick complementarity. The strand labelled $Th_{2,5:5}$ is a "threshold gate"; being more reactive than the other seesaw gates it takes precedence in the reaction. This enables the controlled reaction of gate 5, requiring a presence of signaling molecules beyond the threshold concentration of $Th_{2,5:5}$: A low threshold value causes gate 5 to be activated upon the presence of only one of the inputs, whereas a higher threshold value necessitates the presence of both inputs. Part C shows the required amount of time for the reaction. Taken from [88, Figure 2].*

5.3.2 Seelig et al.'s Gates

The biological Boolean gates presented by Seelig et al. in [100] also rely on DNA hybridization. In their implementation, a gate is a DNA strand with an exposed single-strand, called a toe-hold, that hybridizes with the input, implemented as a single stranded piece of DNA or RNA. This initiates a branch migration, possibly exposing a new toe-hold that matches another input signal (Figure 5.6 part A). The final result of a gate is a single stranded piece of DNA that can react further down the layers of the circuit.

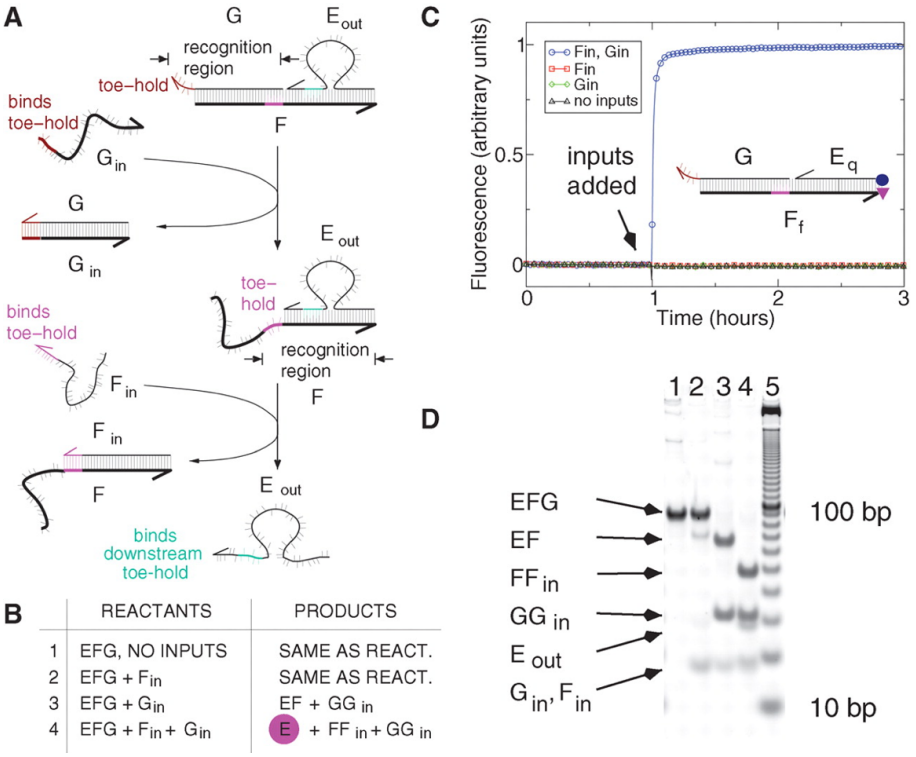
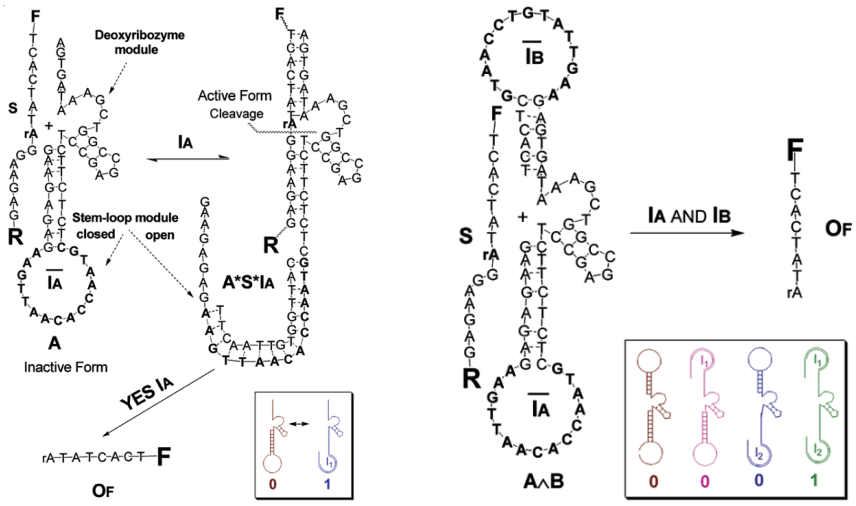


Figure 5.6: *The DNA-based implementation of an AND gate by Seelig et al. [100]. In part A, the DNA structure is represented diagrammatically, revealing the toe-hold domains. Equally colored strands represent Watson–Crick complementary strands, one of them drawn with a bold line and one with a light line. The AND gate requires the presence of G_{in} to expose the purple toe-hold, after which the presence of F_{in} exposes the output E_{out}. Part B is a truth table, and lanes 1-4 of part D corresponds to rows 1-4 in the table. The gel electrophoresis shown in part D confirms that the gate does perform as the truth table indicates, producing E_{out} only in case 4. The illustration is taken from [100, Figure 1].*

The fact that a gate works by "dissolving" itself results in gates being single-use entities. The restrictions regarding the seesaw gates therefore also apply to this kind of gate, again making a hypothetical implementation of the blob model using this type of gate severely limited in its expressiveness.

The time consumption per gate represents a less severe problem with this gate type than with the seesaw gates, although more complex circuits still require several hours to react [100, Figure 3]. In part C of Figure 5.6 the time required for any observable reaction in an AND gate is showed. Only approximately a quarter of an hour passes from the addition of the inputs to a clear fluorescence to be observable, clearly better than the six hours required for the seesaw AND gate (Figure 5.5, part C). The authors note that by using shorter strands throughout the process, faster reaction times should be possible.



(a) *A "YES" gate, implementing the identity function.*

(b) *An AND gate.*

Figure 5.7: *Deoxyribozyme-based gates. The loop structure blocks a certain DNA strand, containing a detectable fluorophore, from attaching itself to the stem of the gate. The input molecule, I_A or I_B, causes the loop to open, revealing a spot for the fluorophore-containing DNA strand. For the AND gate, both input strands have to be present, such that both loops are opened. Illustrations from [106, Figures 3, 5].*

5.3.3 Deoxyribozyme Gates

Stojanovic et al.'s deoxyribozyme-based logic gates consist of specially crafted deoxyribozymes — DNA structures folded into shapes with looped structures [106]. A signal is represented as single stranded DNA, and signal recognition happens when a loop in the gate is unfolded by the signal. The loops contain strands that are Watson–Crick complementary to the inputs, so the presence of an input will unfold a loop.

Using this type of gate it is only possible to create single-layer circuits, due to the fact that input and output signals are different. The output signal is always a specific oligonucleotide containing a fluorophore (labelled "F" in Figure 5.7), making it impossible to discern the output of two different gates. Consequently, one test tube can only contain functions that return a single bit. By using mechanical means such as DNA wells to separate different parts of the circuit from each other, this limitation can be circumvented, as demonstrated for the tic-tac-toe-circuit in [107].

Due to the fact that no means of communication between gates exists, the deoxyribozyme-based gates are impractical for larger circuits with, e.g., 100 components, as the connectivity of the components must be represented by attaching more loops on a deoxyribozyme. This is also the approach taken in [107]. Therefore, this gate is not well-suited for a blob model implementation.

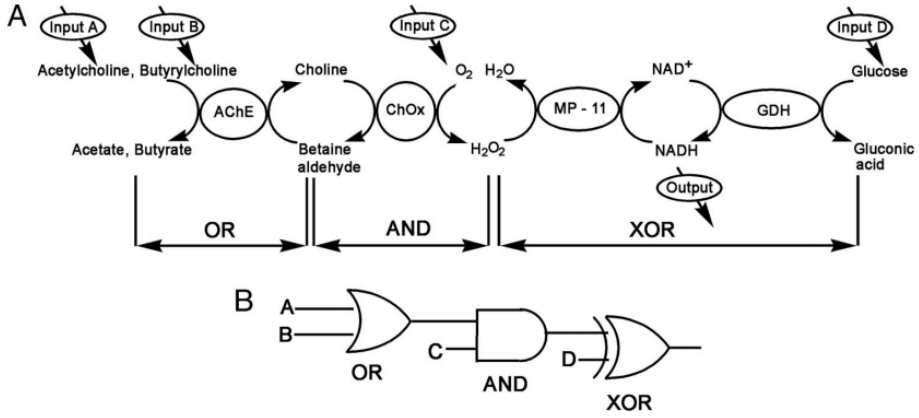


Figure 5.8: *Figure from [78] demonstrating a circuit composed of four different enzymes. The enzyme* AChE *reacts with either of the two inputs (acetylcholine or butyrylcholine), producing choline. Thus, the presence of the product of the reaction can be construed as a logical OR of the two original inputs. This output reacts with the enzyme* ChOx *if a third input, oxygen, is present. Hence, the product of the reaction catalyzed by* ChOx *is a logical AND of the reactants. Finally, the two enzymes* MP-11 *and* GDH *together implement an XOR function: On the absence of input D (glucose) and presence of the output of the AND gate (H_2O_2)* MP-11 *catalyzes an oxidation of $NADH$ to NAD^+; when glucose is present and H_2O_2 is absent,* GDH *catalyzes the reverse reaction, turning NAD^+ into $NADH$. On the presence of both inputs, the reactions cancel each other out, and on the absence of both inputs no reactions occurs. Therefore one can read the output of the XOR function by studying the ratio of NAD^+ to $NADH$.*

5.3.4 Enzyme Gates

The gates introduced in [11] work by exploiting the intrinsic properties of different enzymes, creating an artificial counterpart to what is known as a "pathway" in the cell. The technique is further explored in [78], demonstrating the ability to create layered circuits, see Figure 5.8.

A major problem with this approach is its scalability. No systematic design of signaling molecules is used, as is the case with DNA-based signals. Rather, when designing a circuit using these gates, one is forced to find a set of enzymes that are "compatible" with each other, meaning that the end product of the reaction catalyzed by A is used as the start product of the reaction of enzyme B. In other words, an enzyme type may only be used once in a circuit, assuming that no physical compartments like membranes are used.

Constructing large-scale circuits will quickly deplete the reservoir of available enzymes. The Comprehensive Enzyme Information System [24] lists 5,373 different enzymes (August 2011), so even with the generous assumption that we are able to find a pathway utilizing every enzyme we cannot create a circuit containing more than 5,373 gates. Of course, if all we want is to implement a specific subset, which is to be enclosed in a membrane, the potential size of an

enzymatic circuit grows, as multiple uses of each enzyme would be facilitate by the compartmentalization.

One interesting property of the enzyme-based gates is that they can be made to produce output in electrical form, possibly making it easier to mix traditional circuits with enzyme circuits [135, 66].

5.3.5 Moving the APB-ADB Bond

Three of the gate implementations (the seesaw gates, the Seelig gates, and the deoxyribozyme gates) have the property that a gate is dissolved upon use, making a circuit constructed using them a single-use entity. A circuit representation of the blob model must be able to perform the same computations several times, for instance those associated with the APB-ADB bond. As that is impossible, we cannot change the representation of the bond in a way that emulates the "movement" of the blob model.

The fourth gate implementation, the enzyme-based approach, does not suffer from this limitation.

5.3.6 Cargo Storage

Regarding cargo storage, problems related to the single-use nature of some of the gates makes an implementation of a memory-circuit impossible, as a memory that disappears upon use is rather impractical. The enzymatic gates do not suffer from this problem, and could therefore in principle be used as gates in, e.g., a flip-flop, although we would still face the problem that several unique enzymes have to be used in the implementation.

5.3.7 Implementing `SCG 1 5`

In a hypothetical implementation of the blob model with one of the gate types described here, the subtasks involved in performing one execution of the instruction `SCG 1 5` are as follows:

- The coupling between a blob and a circuit must be made clear. A blob will likely be some sort of membrane enclosing a circuit.
- The APB emits a signal indicating the action that the APB must perform. This signal must only be intercepted by the APB, which is why the implementation relies on some sort of encoding.
- After having received the signal, the APB alters its internal memory state.
- Control is propagated to the next program blob.

Each of these subtasks presents substantial problems in a circuit context. Disregarding the single-use issues, we need a way to implement a memory, a systematic method for encoding signals (which is problematic with the enzymatic gates), and a method for controlling the execution by propagating the "activation bit."

Type	Multi-use	Multilayered	Systematic encoding
Seesaw gates [88, 87]	No	Yes	Yes
Seelig et al. [100]	No	Yes	Yes
Deoxyribozyme [106]	No	No	Yes
Enzyme [11, 78]	Yes	Yes	No

Table 5.2: *Overview of the problems with the different biomolecular Boolean gates.*

5.3.8 Feasibility

The different versions of biological Boolean gates described here present some rather unfortunate obstacles as implementation substrates:

- Some of the gates can only be used once, leading to single-use circuits that cannot implement a blob (but, as noted in Section 3.3, could conceivably be cryptographically interesting).
- Others still are unable to describe layered circuits.
- The only gate type that is capable of both implementing circuits with several layers and being used more than once (the enzyme gates) has the property that no systematic signal encoding is used.

Based on these constraints (outlined in Table 5.2), biological Boolean gates by themselves are not a feasible implementation substrate for the blob model. This observation does not preclude the possibility that biological gates may play a role in an implementation based on other principles as well; the conclusion only concerns "pure" biological Boolean circuits.

5.4 FokI Restriction

Benenson et al.'s DNA automaton based on repeated use of the restriction enzyme *FokI* has computational power equivalent to that of a DFA. Hence, for every regular language one can construct a *FokI*-based DFA recognizing it (in principle, barring physical considerations). As it is not a Turing-universal form of computation, it cannot be used to implement the blob model directly. However, that does not mean that the underlying mechanism in the *FokI*-DFA cannot be used in an implementation of the blob model.

5.4.1 Moving the APB-ADB Bond

It is not clear how to move bonds in a *FokI* implementation of the blob model. Initially, we must attempt to clarify how the implementation is made, if a discussion of bond movement is to be sensible.

Given a method to implement a single blob as a *FokI*-DFA, we still face the problem of how to attach multiple blobs together, not to mention the issue of how to alter these bonds. Simply mixing several DNA strands, each representing an automaton, together in a large pot is likely to introduce a host of problems, due to the inability of the transition molecules to discern between different

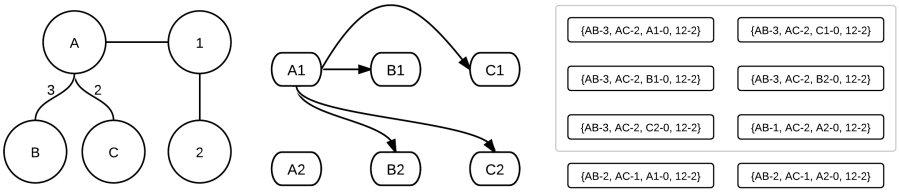


Figure 5.9: *A blob program and two different types of bond automata, encoding the bond state changes. The automaton in (b) encodes only the possible bond transitions, thereby saving space, but it requires one to know the possible program flow in advance. In (c) the automaton encodes all possible bond configurations, i.e., all bonds in the program, giving rise to too many states (only a few are shown). The states inside the grey box correspond to the states in (b). Transition arrows are omitted because they go from any state to any other.*

blobs. A problem similar to that of the localized hybridization of the DNA tiles arises, again stemming from the requirement that only one bond may be active. Thus, we require a method to restrict the execution of our imaginary blob implementation.

5.4.1.1 A Bond Automaton

Maintaining some special automaton, whose sole job is to control the position of the APB-ADB bond, might solve the problem. Each blob pair would be represented as a state of an automaton, and the current state of that automaton controls which blob should be activated. The blobs are each assumed implemented as some larger automaton that are not physically interconnected. Figure 5.9 shows two rough sketches of two different bond automata.

A blob program has an astronomically large number of possible orderings of its bonds. In the most extreme case, where self-bonds are allowed (thus introducing the risk of disjoint parts of a program), the number of possibilities for bond orderings are $\prod_{j=0}^{4n-2} \binom{4n-j}{2} \cdot \frac{1}{2}$ for a program with n blobs. This is observable when thinking about the the bonds as follows: Given n unconnected blobs, we pick two bond sites of the $4n$ possible and add a bond. As a bond has no direction, we have $\binom{4n}{2} \cdot \frac{1}{2}$ possibilities. The next bond must then be placed between two of the $4n-2$ remaining bond sites, ultimately leading to the product stated above. Of course, this is assuming that *every* possible bond will be exhausted, so it is a quite loose upper bound. Still, the size of it is discouraging; although a realistic program will never have exhausted every possible bond site, it may still realistically use three out of four bond sites on almost every blob for a heavily-branching program.

The upper bound translates into an astronomically large number for a reasonably sized blob program. For instance, for 200 blobs the number of possible combinations are in the order of 10^{3710}. Clearly, encoding the entire space of possible bond combinations is infeasible; a DNA strand of length approximately $\log_4(10) \cdot 3710 \approx 6100$ base pairs would be required for each possibility. Part (c) of Figure 5.9 is an illustration of this situation, with only a

(very) small subset of the possible bond combinations drawn.

Instead of encoding elements of the space containing all possible blob configuration vectors, we turn to the encoding scheme in which we equip each blob with a unique identifier and use these to encode bonds. Each bond encoding consists of two blob identifiers and one integer in the range from one to four that indicates the bond site. For a 200-blob program, this encoding requires 9 base pairs on a DNA strand, as the blobs require four base pairs each to uniquely identify them and the integer requires one base pair. Again assuming that we have some implementation of the blobs present, we could implement the bond change using transition molecules that matches a bond-encoding DNA strand and releases a new DNA strand that encodes an altered bond. In part (b) of Figure 5.9 this approach is sketched.

With this bond encoding, we require the presence of (i) the correct transition molecules, and (ii) the knowledge that only the correct one will be applied in order to direct the program flow.

For the former requirement, the implementation must be able to synthesize the transitions dynamically, as the generation of all possible transitions at "compile time" requires a prohibitively large number of different molecules: Representing a bond as described above, we have $(n \cdot (n-1) \cdot 4^2)^2 \approx n^4$ different possible transitions, as there are $n \cdot (n-1) \cdot 4^2$ possible bonds, from each of which a transition to another bond can be made. Naturally, the large majority of these transitions will never be used, as the active program and data blob change in a restricted way, but because bonds can be altered programmatically, we cannot know beforehand which bond transitions will be used. Moreover, adding relative transition molecules like "go to the blob immediately below the APB" does not alleviate the need for a unique representation of the APB. We therefore need the entire set of transition molecules from the beginning.

For the latter requirement, we face the familiar problem of how to control the reaction between the restriction enzyme and the DNA strands, such that it does not run amok. As can be seen, none of the two requirements stated above have straight-forward solutions.

5.4.1.2 Physical Bonds

Moving the focus to an implementation in which blobs are physically adjacent, connected by chemical bonds between DNA strands, we enter problems regarding the distributed nature of DNA reactions in general. If the connection is physical, we are basically in the same situation as we were with the DNA tiles, and solutions must therefore be found to the same set of problems: We must perform some form of local hybridization and a way to permute nucleotides in some way must be present. Furthermore, we also require some form of control over the *FokI* enzymes, such that they only cleave DNA strands were they are supposed to.

5.4.2 Cargo Storage

If we ignore the fact that blobs communicate and that they perform certain operations based on their cargo bit value, instead looking only at the way the

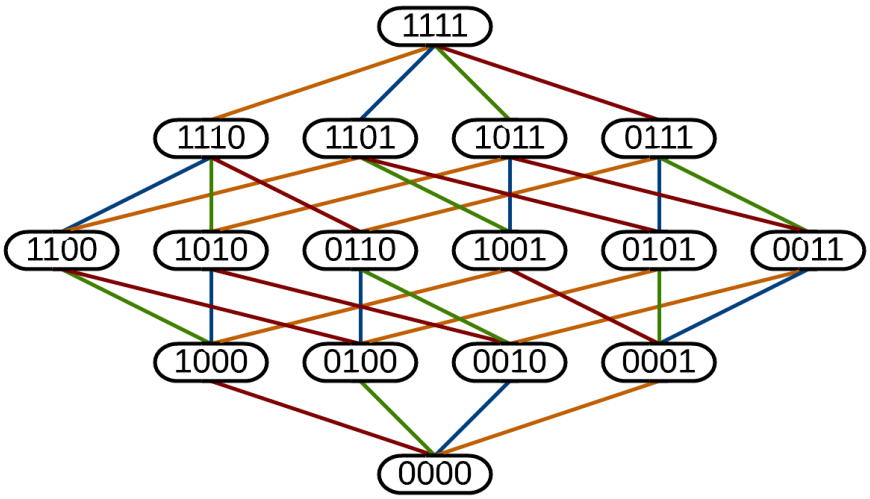


Figure 5.10: *State diagram for an automaton implementing a four-bit memory, which can only be manipulated one bit at a time. Every edge between S and S' represents transitions $S \to S'$ and $S' \to S$ (arrow heads omitted for clarity). The edge coloring represents the different transition symbols: Red swaps bit 1, green bit 2, blue bit 3, and orange bit 4.*

seven* cargo bits are stored and manipulated, we obtain a state transition graph in which each vertex represents a bit configuration and each edge represents the flipping of one bit. In Figure 5.10 such a graph is illustrated for a four-bit memory. With n bits, the graph contains 2^n vertices and $\sum_{i=1}^{n} i\binom{n}{i}$ edges. Hence, for seven bits we need 128 vertices and 448 edges, giving 128 states and, because the transitions have directionality, $2 \cdot 448 = 896$ transitions in the automaton.

A transition is associated with two states and a symbol, representable as the tuple $\langle S, S', c\rangle$. The transition graph of the automaton is regular,† whence we need n distinct symbols to disambiguate the transition table for an n-bit memory. Consequently, the minimum number of bits required to represent a transition molecule is $2 \cdot \log_2(128) + \lceil \log_2(7) \rceil = 17$. A DNA strand with n bases can represent $4^n = 2^{2n}$ different strings, that is, it carries $2n$ bits. The length of the transition molecules must therefore be at least 9. The recognition site for *FokI* is five bases long [17], making the transition molecules 14 base pairs long. Note that this number is the information-theoretical minimum amount required to represent the states and symbols. In reality, auxiliary base pairs will be needed, due to spurious matches between DNA strands. For instance, the implementation of a two-state two-symbol automaton demonstrated in [17] employed transition molecules of length 10, 12, and 14 bases.

The automaton from [17] is only capable of accepting or rejecting an input string. Once in an accepting state, the automaton is "used" and cannot perform more transitions. A DFA that represents memory as described here does not take a whole string as input, rather, it awaits symbols one at a time, each

*The high-order bit indicating the APB is ignored for now.
†Each vertex has the same degree.

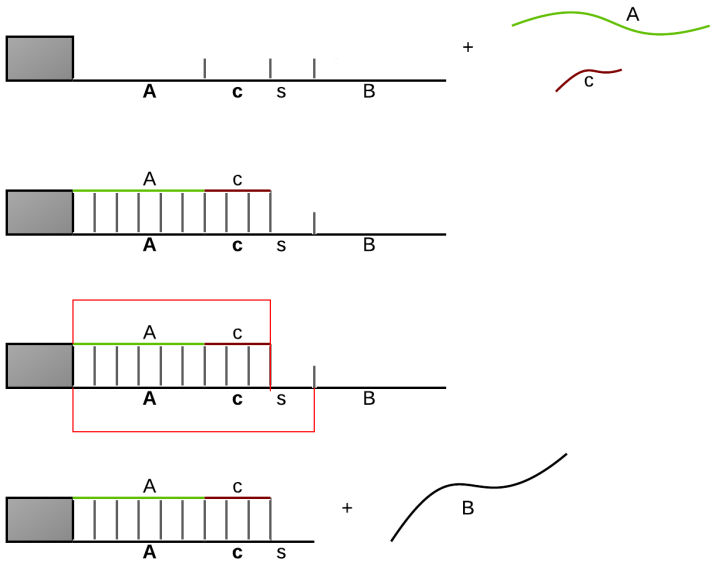


Figure 5.11: *Sketch of a transition from state A to state B upon receiving the symbol c. Bold letters indicate the Watson–Crick complementary strands. Because the* FokI *enzyme cuts the DNA double helix asymmetrically, a spacer, denoted "s," is needed. Contrary to the original approach, illustrated on Figure 2.4, this DFA does not rely on an input string given at the start of computation.*

causing it to do a transition to a new state. None of the states are accepting states, and the automaton must be able to run indefinitely. For instance, the instruction `SCG 1 5` would cause the symbol that sets cargo bit five to the value 1 to be emitted and read by a memory automaton.

As the blob emitting the "set cargo bit n to 1"-command does not know the value of the data blobs memory, the transition symbols must be associated with each bit position. So, the state S having seven incoming and seven outgoing edges in the transition graph would uniquely associate with each symbol a specific transition, and, crucially, the same symbol modifies the same bit position regardless of the contents of the memory. It may be needed, though, to introduce two symbols for each transition edge, such that `SCG 1 5` does not accidentally set bit five to 0 if it is already on. Figure 5.10 illustrates the different symbols with differently colored edges, albeit with the same symbol for both directions of each edge.

5.4.2.1 A Memory Automaton

A potential implementation scheme for the cargo bits is sketched on Figure 5.11. The current state is represented as a DNA oligonucleotide. After the symbol is added, both state and symbol molecule hybridize with the transition molecule which matches that particular state/symbol combination, whereafter the restriction enzyme cuts the DNA strand and a new state molecule is released. *FokI* cuts the helix nine and 13 base pairs removed from the recognition site, so the combined length of the state molecule and the symbol molecule is nine

base pairs, equivalent to 18 bits. This exceeds the number of bits required to represent one state and one symbol in the equations above, so it is enough. However, it might be beneficial to reduce the cargo size to, e.g., six bits, to allow for better encodings that can eliminate spurious DNA matches.

5.4.2.2 Problems

The cargo storage method suggested here is only a sketch. Several issues have to be addressed before it can be ready for an actual test.

- First of all, a problem arises when the DFA has to choose the correct transition molecule: Each molecule is associated with a state/symbol combination, but the state is contained in the DFA prior to the addition of the symbol. Thereby we risk that the state molecule partly hybridizes with some other transition from the same state. If it did so, the correct transition molecule would not be activated.
- To avoid running the risk of hybridizations between memory state DNA strands, we must choose an encoding scheme that minimizes the amount of Watson–Crick complementarity between the states, lowering the number of bits that can be packed into each state molecule. A simple way of doing it is to let every second base be fixed to the same nucleotide and encode the state in the other half of the bases. As all the state molecules would have the same fixed bases, the states would be non-Watson–Crick complement in at least half of their length.
- After a successful transition, the transition molecule is spent and cannot be reused as it is. Accordingly, we either need a method to remove the spent transition molecules to reduce clutter, or we must be able to make them usable again. If the former solution is used, there could lie a problem in that all the transition molecules must be present in "large enough" quantities. Alternatively, new transition molecules could be added manually during the lifetime of the memory automaton.
- The suggested implementation described above focuses only on how to adapt a *FokI*-based DFA to function as a seven-bit memory device. Hence, it does not take into account how to differentiate between different blobs. If each blob is made entirely like this (assuming that a method for controlling the behavior of each blob is found), and if they all exist in the same sphere (test tube, e.g.), they must necessarily either be physically separated in some way, or use unique encodings for states and symbols. The latter is infeasible, as the number of blobs would cause an explosion in the number of bits required to encode the states, far exceeding the 18 bits theoretically possible with the transition molecules.

5.4.3 Implementing `SCG 1 5`

We continue the pattern laid out, using the `SCG 1 5` as a test object of implementation.

1. Acquire the ADB somehow. If the first one of the two discussed methods to manage the APB-ADB bond is used, we either require a reliable method

to let blobs synthesize transition molecules online, or we synthesize all of them a priori, thus requiring a way to ensure that not all of them react simultaneously. If instead the latter of the two mentioned methods is used, a physical bond between the blobs is established, presenting the previously discussed problems related to that.

2. Having established the contact between APB and ADB, the APB sends the signal "5_1" to the active data blob's memory automaton, either using an encoding scheme dictated by the transitions responsible for the bond propagation, or relying on the physical connection between the blobs.

3. The ADB performs the appropriate transition in its memory automaton.

4. The next blob to be the APB is the one connected at bond site "2" of the previous APB. Either we update the state of a bond automaton, or we move a physical bond. None of these are easily done, as reflected in the preceding discussions.

In summary, the issues that face a *FokI*-based implementation can be listed as follows:

- Controlled *FokI*-reactions, so the correct transition molecules are chosen.
- Separation between different blobs.
- Proper handling of spent transition molecules.

5.4.4 Feasibility

Using a *FokI*-based DFA to implement the complete blob model is not likely to be possible, due to the problems illustrated in the preceding discussions. Although, an implementation of the memory part of a blob does seem to hold more promise, as the implementation sketch should illustrate. Further looking at *FokI*-based approaches is therefore not entirely discouraged, as it may be able to play a part in conjunction with other methods.

Finally, it must be noted that while we have focused here entirely on the enzyme *FokI*, several other restriction enzymes with similar capabilities exists. Recall, for instance, that Shoshani et al. used the enzymes *NcoI* and *PstI* to produce phenotypic output in their DFA implementation [102].

6 Future Substrates

As none of the three established methods examined in the previous chapter appears to present a viable implementation substrate for the complete blob model, we turn now to a discussion about a theoretical implementation scheme, one that appears "biological," but without any experimental track record.

In the sketch of this implementation, the blob model is slightly altered, as the three constraints mentioned in the beginning of Chapter 4 are unfulfilled.

6.1 Artificial Cells

Handcrafting artificial cells that represent blobs more directly could be a possible implementation strategy. Such artificial cells would consist of a membrane encapsulating the machinery that makes the blob work. The machinery could contain elements from the previously implemented methods.

6.1.1 Making Blobs Physical

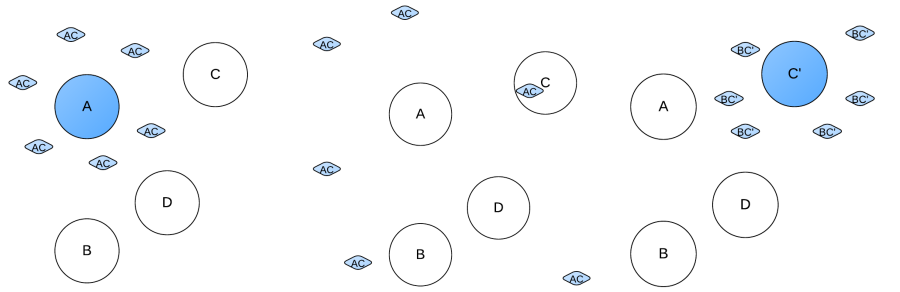
The intuitive interpretation of the blob model, in which blobs are physically connected, need not be the only one. In the following, we will depart from this understanding of the model.

The main idea is to look at the blobs as nodes in a tiny mesh network, in which bonds are represented by signals emitted from each blob. Without violating that analogy, we may add the additional requirements that some of the nodes are "data nodes," representing the data blobs. It is important that they are not implemented differently than the nodes representing the instruction blobs if we are to maintain the hallmark of the blob model: the equivalence of software and hardware.

In such a setup, two different molecule families are present: The *signaling molecules* that play the role of the the signals described above, and the *blob molecules*, that implement the blobs.

Each blob molecule in the network described above must be capable of several different things:

- Receive a signal, i.e., the molecule released by one of the surrounding nodes, and perform some action only if the node matches the signal.
- Store cargo bits in some structure.
- Implement molecular machinery capable of reacting based on the contents of the cargo bits. In other words, something capable of 128 different instructions must be present (the high-order bit is unused in the instruction set).
- Let the behavior of the machinery create a new batch of signal molecules that are introduced to the surroundings.



(a) *Blob A releases a signal.* **(b)** *Blob C reacts.* **(c)** *C is altered to C′, and releases a signal receivable by B.*

Figure 6.1: *Signaling between blobs. The physical bonds are absent, and are instead implicitly contained in the synthesis of the diamond-shaped signaling molecules. Blob A does something with blob C, turning it into C′. Subsequently, blob C′ tells blob B that it has become the APB, an information that may be contained in message AC. This process is repeated over and over, causing B to receive BC′ and release BD, which again causes D to receive BD and release, e.g., D′E.*

6.1.2 Proximity

In order for the signal molecules to be able to propagate properly, the blobs have to maintain some maximum-proximity invariant. If the blobs are distanced too far apart in the solution, there is a risk that the signal molecules either cannot reach their targets, or that they do so too slow. Any blob can communicate with any other blob by properly rearranging their bond information, so all blobs must be packed together closely. Making the blobs stick to each other could solve the problem, as any blob program would clot and form a ball-like structure. The method for gluing together blobs must allow for signaling molecules to pass unhindered between the blobs.

6.1.3 Moving the APB-ADB Bond

The APB-ADB bond cannot be moved explicitly, as it is not physically present in the implementation. Rather, it is a result of the combined behavior of the released signaling molecules. The "movement" of the bond is accomplished by controlling the released signaling molecules such that the correct blob reacts on an released signal.

A strategy for this is to follow a "call-response" pattern, illustrated with Figure 6.1, in which one instruction blob releases a signal at time t_0, followed by a new signal released by the affected data blob at time t_1, which in turn affects the next program blob that releases a new signal at time t_2, and so on. Thus, every signal is either released by an instruction blob and received by a data blob, or released by a data blob and received by an instruction blob, fitting well with the notion that data and instruction blobs are equal.

The call-response signaling strategy introduces the problem of how to discard "used" signals, i.e., signals that have been received and acted upon properly.

The problem is that of n signaling molecules, the $n - 1$ that did not reach their target cannot suddenly self-destruct because one signal was successful in reaching its destination. Assuming that signals always either cause no action, or causes some action at the desired target blob, one could implement a "fence" around the blob program, consisting of reactants that neutralize any signaling molecule. Thereby all signals would be discarded automatically when they got too far away from the program itself.

6.1.3.1 Maintaining a Single APB-ADB Bond

One must be able to guarantee with some reasonable probability that only one blob is reacting at any time, causing no more than one APB-ADB bond to be present. As the activation bit is interpreted as a kind of baton, contained in the signals being passed between blobs, we can reduce the problem to guaranteeing that no more than one blob possesses a baton.

Due to the presence of more than one signaling molecule, a blob will have to wait some amount of time before emitting a new set of signals, to allow for the unused signaling molecules to be removed, or to let them be distanced adequately far away.

If the signaling molecules were designed such that they carry electrical charge, we could force them through the blobs using a technique similar to a gel electrophoresis: After emission, all signaling molecules are led from side to side of the solution using an electrical field in which the anode and cathode are repeatedly interchanged. During this movement, every blob is visited, and the target blob has received its signal. Thus, after the last movement of signaling molecules they can be destroyed. The idea relies upon the blobs being non-charged, or at least that they move sufficiently slow during the electrophoresis.

6.1.4 Cargo Storage

As it was the case with the DNA tile-based solution discussed previously, the cargo bits must be stored in a way that allows the blob behavior to be conditioned on their values.

The cargo can take 128 different values (the activation bit is unused), and a slightly smaller number of instructions must be supported. If each instruction is encoded as a specific biological Boolean circuit, we require 128 circuits. Naturally, this requires the problems listed in the previous discussion of biological Boolean circuits to have been addressed. Hence, 128 distinct molecules are present in each blob, and the blob's cargo value is determined by an "on" marker that resides on exactly one of 128 different molecules.

Another possibility would be to use a *FokI*-based memory automaton, designed in the manner outlined earlier.

6.1.5 Implement `SCG 1 5`

Like before, we study the requirements to implement the `SCG 1 5` instruction, as it is the least complex instruction that performs moderately complicated work. As noted earlier, although the `EXT` instruction is the simplest in the previously described hierarchy described, it does not perform any "real" actions,

as its only role is to halt the entire process.

To implement `SCG 1 5` the following is needed:

1. The blob for `SCG 1 5` releases a number of signaling molecules containing the identifier for the ADB, the identifier for the subsequent APB, and the instruction code.

2. All blobs except the ADB ignores the signal. The ADB consumes it and alters its payload, i.e., moves the "on" marker to another cargo value molecule, or fires a signal to its *FokI*-based memory.

3. The ADB releases a new signaling molecule, targeting the subsequent APB that was obtained from the previous signal.

Accomplishing these tasks poses several challenges to the implementation. The following is an enumeration of some of the problems that would have to be solved, if an implementation of the blob model should be realizable:

- Every blob needs a unique marker that is used in the signaling molecules to target the blob. For large programs, this could be a problem. Furthermore, the number of different markers required also grows in the input size, because the data blobs also need markers.

- Some kind of representation of the action to be performed must be transmitted in the signal molecule.

- Each blob must be encapsulated in a membrane-like structure, allowing only signal molecules with a specific marker to pass through. This resembles the way molecules interact with "real" cells, although we are presented with the problem that each blob is uniquely addressable.

- A set of "memory molecules" must be designed, some of which encodes actions, like setting a bit.

- Synthesis of new signal molecules, conditioned on the output of the acting memory molecule, has to be performed in the blob, and the synthesis must produce an adequate amount of signaling molecules, to ensure that the target blob is hit by one of them.

The benefit of this method is that the reliance on forming and breaking of physical bonds has been removed, causing instructions such as swap to be easier implemented, given that the requirements mentioned above have been met: Swapping bond sites is now a question of equipping blobs with new marker molecules that they attach to their emitted signaling molecules, a situation that is arguably less complicated than the controlled physical displacement of blobs and bonds.

6.1.6 Marker Molecules

Every blob molecule is associated with a unique marker molecule, e.g., a DNA strand containing a unique base pair sequence. However, every blob need not know the markers of all the other blobs, due to the adjacency criteria in the design of the blob model. For any blob, it is enough to only know the markers of the blobs connected to it. For the swap instructions a few more are needed, but still not the entire set of blobs. The set of markers known to a blob must be changeable, as the bond sites may be altered.

New blobs must be associated with markers that are not present in any of the existing blobs. The probability of randomly generating a new, unique DNA strand rises with the length of the strand, so we must choose signaling molecules of sufficient size.

Recall that a DNA strand with n base pairs can store $2n$ bits. Choosing DNA strands with a length of 25 base pairs should therefore give 2^{50} different combinations. Therefore, assuming that marker molecules are recognized perfectly, a blob self-interpreter consisting of 50,000 blobs is able to use (i.e., address) $2^{50} - 50,000 \approx 2^{50}$ blobs, equivalent to a memory of $2^{50} \cdot 7 \approx 896$ TiB. Due to the fact that spurious recognitions, situations where a DNA strand is interpreted as another strand, may appear, some restrictions on the encoding of the DNA strand must be present.

6.1.7 Effectiveness

We make the somewhat strong assumption that a blob is circular, all signaling molecules from a blob are emitted at the same time, and that they travel radially outwards with equal speed, such that they form increasingly larger concentric circles as time passes. Furthermore, we limit ourselves to an argument in only two dimensions.

Let the number of blobs be n and let the diameter of a blob be d. For the sake of the argument we pick one specific blob at random from an imaginary collection of blobs. The maximum distance to another blob is denoted Q. At emission time, the signaling molecules are on a circle with the circumference $\pi \cdot d$, and the average space between each molecule is $b \approx \frac{\pi \cdot d}{n}$. When the circle containing the molecules reaches the farthest blob, the average distance between each signal is $A \approx \frac{\pi \cdot (d+Q)}{n}$. If the signaling molecules are to reach any of the blobs, the spacing between them must be consequently smaller than A.

Now, if we arbitrarily assume that a blob is ten times larger across in this implementation than in the DNA tile-based one, we have $d = 150$ nm. Furthermore, assuming that a blob program consisting of 1000 blobs is contained in a circular area of $1000 \cdot (\pi \cdot 80^2 \text{ nm}) \approx 2 \cdot 10^6 \text{ nm}^2$, with the radius $r = (\frac{2 \cdot 10^6 \text{ nm}}{\pi})^{\frac{1}{2}} \approx 2 \cdot 10^3$ nm, we have that the maximum distance between two blobs is $Q = 4 \cdot 10^3$ nm. Hence, if 100 signaling molecules are emitted, the average distance between each blob must be smaller than approximately 67 nm. Our previous assumption, placing the blobs in a circular area, gave us a spacing of 10 nm.

However, the signaling molecules may not be able to freely penetrate any blob, but rather be forced to route around them. If we for the sake of simplicity imagine that the blobs are arranged in a quadratic lattice, we obtain a sort of *taxicab geometry*, in which every route from A to B is equally long, assuming

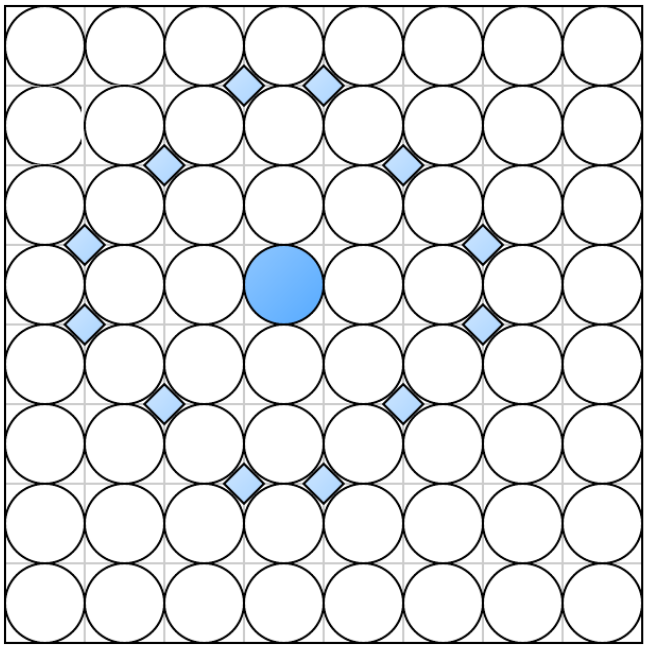


Figure 6.2: *Two-dimensional jellyputer. The blobs (circles) are fixated in a grid, and the signaling molecules can move freely between the blobs. The ADB (dark blue circle) has emitted a set of signaling molecules (light blue diamonds).*

that it does not leave the bounding box defined by A and B, and that it does only move forward. This gives rise to the notion of a "jellyputer," containing the blobs fixated in a gel but letting the signaling molecules move freely. Figure 6.2 is an illustration of such a jellyputer, limited to two dimensions. The idea of fixating larger elements in a grid-like structure is not unfamiliar, as it is also part of the underlying idea in agarose gels used for electrophoresis, although in them the molecules (DNA strands) do move because of the current.

6.1.8 Feasibility

Accurately assessing the feasibility of the approach suggested here is by its imagined nature impossible.

Conclusion

The blob model was described as a "biologically plausible" computational model upon its introduction. After having studied the literature on biomolecular computers, created a formal definition of the "natural programmability" that was the *raison d'être* of the blob model, and chosen six concrete biomolecular methods based on that classification, we are forced to conclude that the biological realizability of the blob model is considerably more difficult than indicated in the description of the model. Hence, an answer to the question formulated at the top of page ix must be in the negative: No, the blob model cannot be realized using the techniques so far used for biomolecular computation.

Based on the classification of computers into either linguistic universality or mechanical universality, three established biomolecular computing techniques were chosen as possible implementation substrates, one of which consisted of four subtypes.

DNA self-assembly. Using self-assembling DNA tiles required us to be able to do the following, none of which is currently possible with the techniques found in biomolecular computing literature:

- perform localized hybridization and denaturation of DNA strands;
- permute nucleotides in a controlled way;
- control the behavior of a DNA tile through cargo bits.

Boolean circuits. The four biological Boolean circuits inspected presented various combinations of the following limitations:

- the gates could be used only once, making it impossible to construct, e.g., a looping structure;
- the gates were unable to be composed into a multilayered circuit;
- the signaling between gates was unsystematic, presenting no simple way to increase the number of gates.

***FokI* restriction.** We described a rudimentary sketch of a way to implement a memory device, but going from there to a blob implementation remains problematic, as we need at least the following:

- signaling, i.e., transition molecules must be constrained to only one DFA, lest different blobs interfere in undesirable ways;
- localized control, ensuring that only one specific DFA acts at any time;
- removal of spent transition molecules;
- description of more than just the memory part.

This result naturally leads us to consider what to do with the blob model. The choice is between:

1. designing an entirely new model, in the hope that it will fare better with respect to implementability;

2. continue searching for a viable implementation substrate for the current incarnation of the blob model;

3. alter the blob model by identifying the weak spots related to its implementation and attempt to remove them.

By altering the blob model we run the risk of removing its Turing-universality entirely, thereby violating a premier design criterion of the model. Nonetheless, we find that option three is the most desirable route to take, as it provides a platform to continue from, instead of throwing everything aboard like in option one. Moreover, if prudence is taken in changing the model, Turing-universality need not disappear from it. Besides, a weaker computational model possessing some of the desirable features of the blob model, like programmability, but implementable, could very well be of use. Option two is not a good route to take: Even though we have not treated every single possibility from the literature, the ones we have treated have revealed problems that are highly likely to reappear in the context of other implementation substrates.

Future Work

Seeing that we probably cannot construct the blob model in any realistic biomolecular substrate, we turn to the question of what to do next. The fundamental question is, whether it is at all possible to achieve what the blob model set out to do, namely the creation of a linguistically universal computer using biomolecular substrates. An answer to this determines the main principal component in the course to take from here on: Firstly, we focus on a situation where the answer is in the negative; secondly, we focus on the situation with a positive answer.

Discarding Universality

Accepting the loss of Turing-universality in the sense aimed at with the blob model, i.e., linguistic universality, leads us to consider weaker computational models and whether there would be any point in exploring them.

Weaken the model. The drug-delivering *FokI*-based DFAs demonstrate an application of weaker computational models. Inspired by this, a next step would be to alter the blob model, weakening its computational power, until a more implementable version of it can be constructed.

- An interesting route is to construct a non-universal domain specific language (DSL) for medicinal purposes, capable of being used as a targeting component for medicine by both specifying and indicating the required preconditions necessary for the medicine's proper usage. Of course, a Turing-complete DSL would be even more interesting, but likely also to be more difficult to build. Specifically, exploration of the relative benefits of regular languages, context-free languages, and context-sensitive languages in a medicinal usage scenario would be required; how strong a computational model do we need in order to be able to do "interesting" things in relation to medicine?
- Leaving the medicinal applications, more knowledge about how "natural" the various language classes are would help in further developments. For instance, is the class of regular languages somehow "more natural" than the context-free languages, in the sense that existing processes in Nature can be said to belong to the class, and if so, why? Including space complexity would be an obvious way to further that discussion.

Focus on mechanical universality. It is possible that we can obtain interesting properties by looking at a version of the blob model that is mechanically universal only. That entails altering the blob model to remove the traits that enable it to be programmable. Given a cheap, systematic method to synthesize these "blob circuits," a shortcut to programming them would be to program the process that synthesizes the blob construction. Doing so would remove the blob model's ability to perform interpretation of programs in the most literal sense, but as that is unnecessary for many applications, it would not be a great loss.

Altering the model to remove programmability from it requires an understanding of which parts of the model it is that provides this feature. As discussed earlier in this thesis, part of the reason appears to be the straight-forward addition of extra computational power, viz., more memory to the system.

Keeping Universality

If we refuse to give up on the original goal of the blob model, we must create a new "biomolecularly plausible" model, only more plausible than the original.

Which "instructions" are required? To program a computer, a series of instructions must be authored somehow. The format of these vary, and each of them represent one "basic" operation of the machine, assuming a reasonable level of abstraction. The ability to perform more than one basic operation is not unique to computers. A simple mechanical contraption with two levers, each one opening a different door with one of them hiding a lion, has at least two basic operations, namely the opening of one or the other door. The operations have hugely different consequences, but obviously this contraption cannot be programmed. There appears to be some sort of "critical mass" of basic operations that must be present in a system before it can be said to be programmable in an intuitive sense.

However, the nature of operations vary considerably, whence it would be desirable to formulate some minimum requirements of operations. Having such a list of requirements would likely also make it easier to study natural processes and determine whether they can be used in a computational scheme. Unfortunately, making such an axiomatization of computability would entail formulating the Church–Turing Thesis in a mathematically rigorous way; a feat that has not been done in the history of computer science. We could limit the admittedly ambitious scope, seeking a formalization of specific types of processes. This is akin to the various formalizations of DNA strand operations that has been done previously, but with a slightly more advanced ambition.

Concrete implementation problems. Regarding the plausibility of a new blob model, we note that two problems recur in the discussion of the blob model's implementability:

- As a consequence of the fact that signals are "floating" around instead of moving in discrete copper lanes, much higher requirements to the sending and receiving of signals exists. Every signal has to encode its receiver, and every possible receiver must be able to discard every signal but the one appointed to it. Furthermore, to ensure that a signal actually reaches its destination, some minimal concentration must be present. We are thus presented with two main problems:
  - a sender has to be able to synthesize signals in large quantities with unique labels that specify their destination;
  - a receiver must be able to quickly sift through a bombardment of signals to find a specific one.

  These two appear to be fundamental problems with most biomolecular computational models, and solving either one would consequently be a

great improvement. There are fundamentally two approaches one could take in an attempt to address the problems:

  - minimize the amount of signal processing required to recognize a message, either by minimizing the number of signals sent or by designing effective ways to discern signals;
  - change the signaling such that it is fixed, thereby eliminating the need for differentiated signaling.

- Cargo bits as they are defined in the blob model appear to fit poorly with the available biomolecular techniques. Of course, the notion of storing bits in some biological material is not at all strange; the genetic code should be sufficient proof thereof. However, the problem with the cargo bits in the blob model seems to be their ability to determine the behavior of the blob itself, thereby serving both as the memory storage and the actor. We do need a memory of some sort, and a memory that can be read from and written to repeatedly to boot, but it could also be contained in the ordering of the bond sites on the blobs instead of in an explicit cargo bit "module" inside each blob.

  Exploring a version of the blob model in which the "cargo" of a blob was an inherent, immutable property of it would remove the problem of how to implement a memory device, at the cost of introducing the problem of how to effectively represent memory change by alteration of the bond configuration.

Simplify the instruction set. In the treatment of the various implementation substrate candidates, we have limited the discussion to the instruction `SCG 1 5`, based on the reasoning that it is one of the simplest instructions, and the successful implementation of which the realizability of the more complex instructions rely. Noting that the prospects for implementing `SCG 1 5` were rather poor, we must conclude that the situation for the more complex instructions is worse. Even though the authors of the blob model claim that the proposed instruction set is parsimonious, further research into reducing the number of "weird" instructions would improve the chances for the model being implementable in a biomolecular substrate. Especially the various versions of the "swap" appear to be able to cause problems regarding implementability, as they perform rather complicated operations. Removing those instructions, we risk the breakage the blob model's Turing-universality, and we are sure to break the self-interpreter that has been constructed, due to its reliance on swap instructions. However, it is not clear whether it is possible to construct another version of a self-interpreter that does not utilize swap instructions.

Beyond the Blob Model

Strand-Based computing. In the biomolecular world, strand-like structures are found in many contexts: Polysaccharides, fatty acids, DNA/RNA, and proteins are all examples. Being different, they still share the property that they are all composed of simpler molecules attached in a chain-like fashion, albeit the chain may be twisted (DNA) or folded in a variety of shapes (proteins). Seeing this, a fitting question to ponder is whether this abundance of strands could

be utilized in a new biologically plausible computational model. Additionally, designing a grammar of strands such that a special family of strand combinations has a well-defined catalytic property while the substrands themselves also encode data would be interesting. Enabling computations by harnessing primitive folding patterns in, e.g., chains of amino acids would be very interesting, not least because the extension for a "natural interface" in relation to medicine would be obvious, as the result of a computation would basically be a protein.

Promising routes. Lastly, we present a short, fuzzy list of possible routes to take in future research and areas that we imagine will benefit from a biological computing paradigm. The list is descendingly sorted on a (very) loose estimation of the expenses associated with each item. The estimates are based around the idea that the "wetter" routes are the more expensive.

- For the majority of people, one of the most interesting application areas of biological computing techniques is probably the development of medicine. Even without Turing-universality there are likely to be exciting possibilities: The use of DFAs to control drug release is a prominent example of this.
- "Smart materials," like computing concrete,* able to alter internal properties as a result of decisions stemming from minuscule computational elements embedded in it, could be very useful. A wild example: Imagine that concrete had the ability change shape dynamically, altering its fundamental structure based on radio signals emitted by a central controller. The computing devices in the concrete react to the radio signals and performs a transformation of the molecular structure of the concrete as a response, ultimately deforming the concrete.
- Supposing one had a viable and implemented biological computational paradigm in conjunction with an efficient method of letting that device communicate with a classic computer, one would get the best from both worlds: the "natural interface" of the biological device and the speed of the electronic computer. The construction of research equipment for areas like biotechnological and medical research would be an obvious use for this.
- The development of biological models for computation need not be used solely for the purpose of actually constructing such a computer. The ability to make a clear connection between data and program in the manner aimed at with the blob model could be useful in non-biological contexts. For example, scenarios in which no single point of failure may exist could possibly benefit from a blob-based approach, perhaps in concert with distributed computing mechanisms.
- Finally, introducing programmability in a biological implementation requires the study of the relationship between programs, data, and the "executing" machines. Determining these relationships exactly is interesting by itself, as it is both a difficult problem as well as being fundamental to computers in general — biological implementation or not.

*Hereby baptized "comprete."

Bibliography

[1] ACM. Computer Science Curriculum 2008. `http://www.acm.org/education/curricula/ComputerScience2008.pdf`, December 2008. [Online; accessed 15-August-2011].

[2] A. Adamatzky, B. De Lacy Costello, and T. Shirakawa. Universal computation with limited resources: Belousov-Zhabotinsky and Physarum computers. *International Journal of Bifurcation and Chaos*, 18(8):2373–2389, Nov. 2008.

[3] L. M. Adleman. Molecular Computation of Solutions to Combinatorial Problems. *Science (New York, N.Y.)*, 266(5187):1021–1024, November 1994.

[4] B. Alberts, D. Bray, K. Hopkin, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Essential Cell Biology*. Garland Science, Taylor and Francis Group, LLC, third edition, 2010.

[5] M. Amos and P. E. Dunne. DNA Simulation of Boolean Circuits. Technical report, Proceedings of 3rd Annual Genetic Programming Conference, 1997.

[6] I. Ardelean. Molecular Biology of Bacteria and Its Relevance for P Systems. In G. Păun, G. Rozenberg, A. Salomaa, and C. Zandron, editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2003.

[7] I. I. Ardelean, R. Ceterchi, and A. I. Tomescu. Sorting with P Systems: A Biological Perspective. *Romanian Journal of Information Science and Technology*, 11(3):243–252, 2008.

[8] T. Bäck, D. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. Oxford University Press and Institute of Physics Publishing, New York and Bristol, 1997.

[9] R. Bar-Ziv, T. Tlusty, and A. Libchaber. Protein-DNA computation by stochastic assembly cascade. *Proceedings of the National Academy of Sciences of the United States of America*, 99(18):11589–11592, 2002.

[10] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland Pub., 1984.

[11] R. Baron, O. Lioubashevski, E. Katz, T. Niazov, and I. Willner. Logic Gates and Elementary Computing by Enzymes. *The Journal of Physical Chemistry A*, 110(27):8548–8553, 2006. PMID: 16821840.

[12] J. Baumgardner, K. Acker, O. Adefuye, S. Crowley, W. DeLoache, J. Dickson, L. Heard, A. Martens, N. Morton, M. Ritter, A. Shoecraft, J. Treece, M. Unzicker, A. Valencia, M. Waters, A. M. Campbell, L. Heyer, J. Poet, and T. Eckdahl. Solving a Hamiltonian Path Problem with a Bacterial Computer. *Journal of Biological Engineering*, 3(1), 2009.

[13] Y. Benenson. Biocomputers: From Test Tubes to Live Cells. *Molecular BioSystems*, 5:675–685, 2009.

[14] Y. Benenson. RNA-based computation in live cells. *Current Opinion in Biotechnology*, 20(4):471–478, 2009. Protein technologies / Systems and synthetic biology.

[15] Y. Benenson, R. Adar, T. Paz-Elizur, Z. Livneh, and E. Shapiro. DNA molecule provides a computing machine with both data and fuel. *Proceedings of the National Academy of Sciences of the United States of America*, 100(5):2191–2196, 2003.

[16] Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, and E. Shapiro. An autonomous molecular computer for logical control of gene expression. *Nature*, 429(6990):423–429, 05 2004.

[17] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, and E. Shapiro. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(6862):430–434, 11 2001.

[18] G. Beni. The concept of cellular robotic system. *Proceedings of the IEEE International Symposium on Intelligent Control*, pages 57–62, August 1988.

[19] C. H. Bennett. The thermodynamics of computation—a review. *International Journal of Theoretical Physics*, 21:905–940, 1982. 10.1007/BF02084158.

[20] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for your Mathematical Games*, volume 2, chapter 25. Academic Press, 1982.

[21] H.-G. Beyer. *The Theory of Evolution Strategies*. Springer, 2001.

[22] D. Boneh, C. Dunworth, R. J. Lipton, and J. Sgall. On the computational power of DNA. *Discrete Applied Mathematics*, 71(1-3):79–94, 1996.

[23] R. S. Braich, N. Chelyapof, C. Johnson, P. W. K. Rothemund, and L. Adleman. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296:499–502, 2002.

[24] T. U. Braunschweig. BRENDA — The Comprehensive Enzyme Information System. `http://www.brenda-enzymes.org/index.php4?page=information/all_enzymes.php4`, August 2011. [Online; accessed 15-August-2011].

[25] A. Brenneman and A. E. Condon. Strand Design for Bio-Molecular Computation. *Theoretical Computer Science*, 287:39–58, 2001.

[26] A. G. Bromley. Charles Babbage's Analytical Engine, 1838. *IEEE Annals of the History of Computing*, 4(3):196–217, July 1982.

[27] L. Cardelli. Strand Algebras for DNA Computing. In R. Deaton and A. Suyama, editors, *DNA Computing and Molecular Programming*, volume 5877 of *Lecture Notes in Computer Science*, pages 12–24. Springer Berlin / Heidelberg, 2009.

[28] W.-L. Chang, M. S.-H. Ho, and M. Guo. Molecular solutions for the subset-sum problem on DNA-based supercomputing. *Biosystems*, 73(2):117–130, 2004.

[29] B. Chazelle. Natural algorithms. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 422–431, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[30] B. Chazelle and L. Monier. Unbounded Hardware is Equivalent to Deterministic Turing Machines. *Theoretical Computer Science*, 24(2):123 – 130, 1983.

[31] J. P. L. Cox. Long-term data storage in DNA. *Trends in Biotechnology*, 19(7):247–250, 2001.

[32] E. Csuhaj-Varjú, R. Freund, L. Kari, and G. Paun. DNA computing based on splicing: universality results. In L. Hunter and T. Klein, editors, *Pacific Symposium on Biocomputing*, pages 179–190, 1996.

[33] V. Danos and C. Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004. Computational Systems Biology.

[34] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. The Modern library, New York, 1936 (1859).

[35] E. A. Davidson and A. D. Ellington. Synthetic RNA circuits. *Nature Chemical Biology*, 3(1):23–28, 01 2007.

[36] L. N. de Castro. Fundamentals of Natural Computing: An Overview. *Physics of Life Reviews*, 4(1):1–36, 2007.

[37] R. Deaton, M. Garzon, R. C. Murphy, J. A. Rose, D. R. Franceschetti, and S. E. Stevens. Reliability and Efficiency of a DNA-Based Computation. *Physical Review Letters*, 80(2):417–420, January 1998.

[38] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 26(1):29–41, February 1996.

[39] J. Engelfriet and G. Rozenberg. Fixed Point Languages, Equality Languages, and Representation of Recursively Enumerable Languages. *Journal of the ACM*, 27:499–518, July 1980.

[40] D. Faulhammer, A. R. Cukras, R. J. Lipton, and L. F. Landweber. Molecular computation: RNA solutions to chess problems. *Proceedings of the National Academy of Sciences of the United States of America*, 97(4):1385–1389, 2000.

[41] U. Feldkamp, W. Banzhaf, and H. Rauhe. A DNA Sequence Compiler. In A. Condon and G. Rozenberg, editors, *Preliminary Proceedings of the Sixth DIMACS Workshop on DNA Computing*, page 253, Leiden, June 2000.

[42] D. B. Fogel, editor. *Evolutionary Computation: The Fossil Record*. The IEEE Press, 1998.

[43] W. Fontana and L. Buss. The barrier of objects: From dynamical systems to bounded organizations. Technical report, Working Papers WP 96-27, International Institute for Applied Systems Analysis, March 1996.

[44] E. Fredkin and T. Toffoli. Conservative Logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[45] A. G. Frutos, Q. Liu, A. J. Thiel, A. M. W. Sanner, A. E. Condon, L. M. Smith, and R. M. Corn. Demonstration of a word design strategy for DNA computing on surfaces. *Nucleic Acids Research*, 25(23):4748–4757, 1997.

[46] P. Fu. Biomolecular computing: Is it ready to take off? *Biotechnology Journal*, 2(1):91–101, 2007.

[47] V. Geffert. Normal forms for phrase-structure grammars. *RAIRO. Theoretical Informatics and Applications*, 25:473–496, 1991.

[48] S. Goss, S. Aron, and J. L. Deneubourg. Self Organized Shortcuts in the Argentine Ants. *Naturwissenschaften*, 76(579–81), 1989.

[49] F. Guarnieri, M. Fliss, and C. Bancroft. Making DNA Add. *Science*, 273(5272):220–223, 1996.

[50] M. Hagiya. From molecular computing to molecular programming. In A. Condon and G. Rozenberg, editors, *DNA Computing*, volume 2054 of *Lecture Notes in Computer Science*, pages 89–102. Springer Berlin / Heidelberg, 2001.

[51] M. Hagiya, M. Arita, D. Kiga, K. Sakamoto, and S. Yokoyama. Towards Parallel Evaluation and Learning of Boolean μ-Formulas with Molecules. In H. Rubin and D. Wood, editors, *DNA Based Computers III*, volume 48 of *DIMACS Series in Discrete Mathematics*, pages 57–72, Providence, RI, 2000. American Mathematical Society.

[52] M. Hagiya, S. Kobayashi, K. Komiya, F. Tanaka, and T. Yokomori. *The Handbook of Natural Computing*, volume To Appear, chapter Molecular Computing Machineries – Computing Models and Wet Implementations. Springer, 2010.

[53] J. Hartmanis. On the Weight of Computation. *Bulletin of the EATCS*, 55:136–138, February 1995.

[54] L. Hartmann, N. D. Jones, and J. G. Simonsen. Programming in Biomolecular Computation. *Electronic Notes in Theoretical Computer Science*, 268:97 – 114, 2010. Proceedings of the 1st International Workshop on Interactions between Computer Science and Biology (CS2Bio'10).

[55] K. Haynes, M. Broderick, A. Brown, T. Butner, J. Dickson, W. L. Harden, L. Heard, E. Jessen, K. Malloy, B. Ogden, S. Rosemond, S. Simpson, E. Zwack, A. M. Campbell, T. Eckdahl, L. Heyer, and J. Poet. Engineering bacteria to solve the burnt pancake problem. *Journal of Biological Engineering*, 2(1), 2008.

[56] T. Head. Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49:737–759, 1987. 10.1007/BF02481771.

[57] C. Heitsch, A. Condon, and H. Hoos. From RNA Secondary Structure to Coding Theory: A Combinatorial Approach. In M. Hagiya and A. Ohuchi, editors, *DNA Computing*, volume 2568 of *Lecture Notes in Computer Science*, pages 215–228. Springer Berlin / Heidelberg, 2003.

[58] S. Hiroyuki and K. Susumu. New restriction endonucleases from Flavobacterium okeanokoites (FokI) and Micrococcus luteus (MluI). *Gene*, 16(1-3):73 – 78, 1981.

[59] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[60] F. Hollandt. TikZ RNA Codons Example. `http://www.texample.net/tikz/examples/rna-codons-table/`, August 2009. [Online; accessed 30-August-2011].

[61] A. Horn. On Sentences Which are True of Direct Unions of Algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[62] F. J. Isaacs, D. J. Dwyer, and J. J. Collins. RNA synthetic biology. *Nature Biotechnology*, 24(5):545–554, 05 2006.

[63] L. Kari and G. Rozenberg. The Many Facets of Natural Computing. *Communications of the ACM*, 51:72–83, October 2008.

[64] R. M. Karp, C. Kenyon, and O. Waarts. Error-resilient DNA computation. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '96, pages 458–467, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[65] S. Kashiwamura, M. Yamamoto, A. Kameda, T. Shiba, and A. Ohuchi. Potential for enlarging DNA memory: the validity of experimental operations of scaled-up nested primer molecular memory. *Biosystems*, 80(1):99–112, 2005.

[66] E. Katz. Bioelectronic Devices Controlled by Biocomputing Systems. *Israel Journal of Chemistry*, 51(1):132–140, 2011.

[67] S. Kobayashi. Horn Clause Computation with DNA Molecules. *Journal of Combinatorial Optimization*, 3:277–299, 1999. 10.1023/A:1009893911892.

[68] K. Komiya, K. Sakamoto, A. Kameda, M. Yamamoto, A. Ohuchi, D. Kiga, S. Yokoyama, and M. Hagiya. DNA polymerase programmed with a hairpin DNA incorporates a multiple-instruction architecture into molecular computing. *Biosystems*, 83(1):18–25, 2006.

[69] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[70] R. J. Lipton. DNA Solution of Hard Computational Problems. *Science*, 268:542–545, Apr. 1995.

[71] Q. Liu, L. Wang, A. G. Frutos, A. E. Condon, R. M. Corn, and L. M. Smith. DNA computing on surfaces. *Nature*, 403(6766):175–179, 01 2000.

[72] Y. Liu, J. Xu, L. Pan, and S. Wang. DNA Solution of a Graph Coloring Problem. *Journal of Chemical Information and Computer Sciences*, 42(3):524–528, 2002.

[73] C. Mao, T. H. LaBean, J. H. Reif, and N. C. Seeman. Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. *Nature*, 407(6803):493–496, 09 2000.

[74] G. Mauri and C. Ferretti. Word Design for Molecular Computing: A Survey. In J. Chen and J. Reif, editors, *DNA Computing*, volume 2943 of *Lecture Notes in Computer Science*, pages 1986–1986. Springer Berlin / Heidelberg, 2004.

[75] W. McCulloch and W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biology*, 5:115–133, 1943. 10.1007/BF02478259.

[76] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[77] T. Nakagaki. Smart behavior of true slime mold in a labyrinth. *Research in Microbiology*, 152(9):767–770, 2001.

[78] T. Niazov, R. Baron, E. Katz, O. Lioubashevski, and I. Willner. Concatenated logic gates using four coupled biocatalysts operating in series. *Proceedings of the National Academy of Sciences*, 103(46):17160–17163, 2006.

[79] A. Nishikawa and M. Hagiya. Towards a system for simulating DNA computing with whiplash PCR. In *Proceedings of the 1999 Congress on Evolutionary Computation*, 1999.

[80] V. Norris, A. Zemirline, P. Amar, P. Ballet, E. B. Jacob, G. Bernot, G. Beslon, E. Fanchon, J.-L. Giavitto, N. Glade, P. Greussay, Y. Grondin, J. A. Foster, G. Hutzler, F. Képès, O. Michel, G. Misevic, F. Molina, J. Signorini, P. Stano, and A. Thierry. From bioputing to bactoputing: computing with bacteria. pages 123–150, 2008.

[81] M. Ogihara and A. Ray. Simulating Boolean Circuits on a DNA Computer. Technical Report 631, University of Rochester, August 1996.

[82] M. Ogihara and A. Ray. Simulating Boolean Circuits on a DNA Computer. *Algorithmica*, 25:239–250, 1999. 10.1007/PL00008276.

[83] Q. Ouyang, P. D. Kaplan, S. Liu, and A. Libchaber. DNA Solution of the Maximal Clique Problem. *Science*, 278:446–449, 1997.

[84] G. Păun. Computing with Membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[85] G. Păun and G. Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287:73–100, September 2002.

[86] N. Pippenger. Developments in the Synthesis of Reliable Organisms from Unreliable Components. *Proceedings of Symposia in Pure Mathematics*, 50:311–324, 1990.

[87] L. Qian and E. Winfree. A simple DNA gate motif for synthesizing large-scale circuits. *Journal of The Royal Society Interface*, 2011.

[88] L. Qian and E. Winfree. Scaling Up Digital Circuit Computation with DNA Strand Displacement Cascades. *Science*, 332(6034):1196–1201, 2011.

[89] T. Ran, S. Kaplan, and E. Shapiro. Molecular implementation of simple logic programs. *Nature Nanotechnology*, 4(10):642–648, 10 2009.

[90] A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. BioAmbients: an abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, 2004. Computational Systems Biology.

[91] A. Regev and E. Shapiro. Cellular abstractions: Cells as computation. *Nature*, 419(6905):343–343, 09 2002.

[92] A. Regev, W. Silverman, and E. Y. Shapiro. Representation and simulation of biochemical processes using the π-calculus process algebra. In *Pacific Symposium on Biocomputing*, volume 6, pages 459–470, 2001.

[93] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 25–34, New York, NY, USA, 1987. ACM.

[94] K. Rinaudo, L. Bleris, R. Maddamsetti, S. Subramanian, R. Weiss, and Y. Benenson. A universal RNAi-based logic evaluator that operates in mammalian cells. *Nat Biotech*, 25(7):795–801, 07 2007.

[95] R. M. Robinson. Undecidability and nonperiodicity for tilings of the plane. *Inventiones Mathematicae*, 12:177–209, 1971. 10.1007/BF01418780.

[96] S. Roweis, E. Winfree, R. Burgoyne, N. V. Chelyapov, M. F. Goodman, P. W. K. Rothemund, and L. M. Adleman. A Sticker-Based Model for DNA Computation. *Journal of Computational Biology*, 5(4):615–629, 1998.

[97] A. J. Ruben and L. F. Landweber. The past, present and future of molecular computing. *Nature Reviews Molecular Cell Biology*, 1(1):69–72, 10 2000.

[98] K. Sakamoto, H. Gouzu, K. Komiya, D. Kiga, S. Yokoyama, T. Yokomori, and M. Hagiya. Molecular Computation by DNA Hairpin Formation. *Science*, 288(5469):1223–1226, 2000.

[99] K. Sakamoto, D. Kiga, K. Komiya, H. Gouzu, S. Yokoyama, S. Ikeda, H. Sugiyama, and M. Hagiya. State transitions by molecules. *Biosystems*, 52(1-3):81 – 91, 1999.

[100] G. Seelig, D. Soloveichik, D. Y. Zhang, and E. Winfree. Enzyme-Free Nucleic Acid Logic Circuits. *Science*, 314(5805):1585–1588, 2006.

[101] S. Shoshani, T. Ratner, R. Piran, and E. Keinan. Biologically Relevant Molecular Finite Automata. *Israel Journal of Chemistry*, 51(1):67–86, 2011.

[102] S. Shoshani, S. Wolf, and E. Keinan. Molecular computing with plant cell phenotype serving as quality controlled output. *Molecular BioSystems*, 7:1113–1120, 2011.

[103] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, 2 edition, Feb. 2005.

[104] L. M. Smith, R. M. Corn, A. E. Condon, M. G. Lagally, A. G. Frutos, Q. Liu, and A. J. Thiel. A Surface-Based Approach to DNA Computation. *Journal of Computational Biology*, 5(2):255–267, 1998.

[105] D. Sprinzak and M. B. Elowitz. Reconstruction of genetic circuits. *Nature*, 438(7067):443–448, 11 2005.

[106] M. N. Stojanovic, T. E. Mitchell, and D. Stefanovic. Deoxyribozyme-Based Logic Gates. *Journal of the American Chemical Society*, 124(14):3555–3561, 2002.

[107] M. N. Stojanovic and D. Stefanovic. A deoxyribozyme-based molecular automaton. *Nature Biotechnology*, 21(9):1069–1074, 09 2003.

[108] M. Ströck. DNA Double Helix. `http://commons.wikimedia.org/w/index.php?title=File:DNA_Overview.png&oldid=33168806`, February 2006.

[109] X. Su and L. M. Smith. Demonstration of a universal surface DNA computer. *Nucleic Acids Research*, 32(10):3115–3123, 2004.

[110] N. Sudarsan, M. C. Hammond, K. F. Block, R. Welz, J. E. Barrick, A. Roth, and R. R. Breaker. Tandem Riboswitch Architectures Exhibit Complex Gene Control Functions. *Science*, 314(5797):300–304, 2006.

[111] C. Tan, H. Song, J. Niemi, and L. You. A synthetic biology challenge: making cells compute. *Molecular BioSystems*, 3:343–353, 2007.

[112] S. W. Thomas, R. C. Chiechi, C. N. LaFratta, M. R. Webb, A. Lee, B. J. Wiley, M. R. Zakin, D. R. Walt, and G. M. Whitesides. Infochemistry and infofuses for the chemical storage and transmission of coded information. *Proceedings of the National Academy of Sciences*, 106(23):9147–9150, 2009.

[113] H. Uejima, M. Hagiya, and S. Kobayashi. Horn Clause Computation by Self-assembly of DNA Molecules. In N. Jonoska and N. Seeman, editors, *DNA Computing*, volume 2340 of *Lecture Notes in Computer Science*, pages 308–320. Springer Berlin / Heidelberg, 2002.

[114] R. Unger and J. Moult. Towards computing with proteins. *Proteins: Structure, Function, and Bioinformatics*, 63(1):53–64, 2006.

[115] J. von Neumann. *The Computer and the Brain*. Yale University Press, 1958.

[116] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[117] S. Vrist. A Programming Language for a Biomolecular Computational Model. Master's thesis, Department of Computer Science, University of Copenhagen, Copenhagen, October 2010.

[118] S. B. Vrist. Visualizing Blobs and Computation in a Biomolecular Computation Model. 15 ECTS Master's Project, Department of Computer Science, University of Copenhagen, http://blobvis.appspot.com, May 2010.

[119] H. Wang. Proving theorems by pattern recognition. *Bell System Technical Journal*, 40:1–42, 1961.

[120] S. Wang and A. Yang. DNA solution of integer linear programming. *Applied Mathematics and Computation*, 170(1):626–632, 2005.

[121] I. Wegener. *The Complexity of Boolean Functions*. Wiley-Teubner Series in Computer Science. John Wiley and Sons Ltd. and B. G. Teubner, Stuttgart, 1987.

[122] R. Weiss. Challenges and Opportunities in Programming Living Cells. *The Bridge*, 33(4):39–46, 2003.

[123] R. Weiss, S. Basu, S. Hooshangi, A. Kalmbach, D. Karig, R. Mehreja, and I. Netravali. Genetic circuit building blocks for cellular computation, communications, and signal processing. *Natural Computing*, 2:47–84, 2003. 10.1023/A:1023307812034.

[124] R. Weiss, G. Homsy, and T. Knight. Toward *in vivo* Digital Circuits. In *DIMACS Workshop on Evolution as Computation*, Princeton, N.J., January 1999.

[125] Wikipedia. Wang tile — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Wang_tile&oldid=412226388`, 2011. [Online; accessed 16-February-2011].

[126] M. N. Win and C. D. Smolke. Higher-Order Cellular Information Processing with Synthetic RNA Devices. *Science*, 322(5900):456–460, 2008.

[127] E. Winfree. *Algorithmic self-assembly of DNA*. PhD thesis, California Institute of Technology, 1998.

[128] E. Winfree, T. Eng, and G. Rozenberg. String tile models for DNA computing by self-assembly. In A. Condon and G. Rozenberg, editors, *DNA Computing*, volume 2054 of *Lecture Notes in Computer Science*, pages 63–88. Springer Berlin / Heidelberg, 2001.

[129] E. Winfree, F. Liu, L. A. Wentzler, and N. C. Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394:539–544, 1998.

[130] E. Winfree, X. Yang, and N. C. Seeman. Universal Computation via Self-assembly of DNA: Some Theory and Experiments. *DNA Based Computers II, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 44:191–213, 1999.

[131] P. Yin, H. M. T. Choi, C. R. Calvert, and N. A. Pierce. Programming biomolecular self-assembly pathways. *Nature*, 451(7176):318–322, 01 2008.

[132] Y. Yokobayashi, R. Weiss, and F. H. Arnold. Directed evolution of a genetic circuit. *Proceedings of the National Academy of Sciences of the United States of America*, 99(26):16587–16591, 2002.

[133] T. Yokomori. YAC: Yet Another Computation Model of Self-Assembly. In *In Proceedings of the 5th DIMACS Workshop on DNA Based Computers*, pages 153–167, 1999.

[134] T. Yokomori and S. Kobayashi. DNA-EC: A model of DNA-computing based on equality checking. *3rd DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 48:347–360, 1999.

[135] M. Zayats, E. Katz, R. Baron, and I. Willner. Reconstitution of Apo-Glucose Dehydrogenase on Pyrroloquinoline Quinone-Functionalized Au Nanoparticles Yields an Electrically Contacted Biocatalyst. *Journal of the American Chemical Society*, 127(35):12400–12406, 2005.