Computation Model for Clocked DNA Programs

Nimrod Priell

New York University, Courant Institute of Mathematical Sciences
, 251 Mercer Street, New York, NY 10012, May 2012

A thesis submitted in partial fulfillment of the requirements for the degree of master's of science
Department of Mathematics
New York University
05/2012

The author thanks his advisor, Prof. Dennis Shasha for his counsel.

Abstract. We describe a model of computation similar to Probabilistic Finite Automata in which computation is carried out solely using toehold-mediated strand displacement kinetics in DNA complexes. The model's computation proceeds autonomously by repetitively pouring in just one mixture of "clock" strands, and exhibits both synchronized and asynchronous (parallel) computation. We show that this model is capable of accurately solving problems like deciding if an input DNA contains certain sequences, the choice of sequences determined by a boolean logic formulae, or adding two binary-encoded numbers. Furthermore, we have developed code to "compile" an abstract description of the constituent DNA domains in a mixture to the exact sequences and complexes required to carry out the computation, as well as simulate the progression of the experiment, so verification is possible prior to executing the actual computation in a test tube.

Contents

Preface vii

Chapter 1. Introduction 1
- 1.1. Clocked DNA Computing 1
- 1.2. Gated stacks 9
- 1.3. Existing work 10

Chapter 2. Chemical underpinnings 13
- 2.1. Introduction 13
- 2.2. Thermodynamics 13
- 2.3. Kinetics 16

Chapter 3. Probabilistic Automaton Model of Clocked DNA Computing 21
- 3.1. Introduction and existing results 21
- 3.2. Clocked DNA computation as an automaton 21

Chapter 4. Simulation algorithms 31
- 4.1. Implementation of a compiler 31
- 4.2. Algorithms for describing the state space of a program 32

Chapter 5. Results and example clocked DNA programs 37
- 5.1. Strand recognition 37
- 5.2. Digital Electronic Logic Circuit Analogy 51
- 5.3. Future research directions 51

Bibliography 53

Preface

The interest in chemical computation stems from its intellectual appeal and practical potential. As a conceptual framework, many models of computation, notions of computability and languages for expressing algorithms ranging from the well-known such as Automata and Turing Machines to the less familiar Petri Nets, Vector Addition Systems, General Recursive Functions and so on have been explored. As is common in algebraic research, it is the equivalence between them that often highlights properties of both that are less obvious when inspecting just one or the other. As the inspiration for each model is often rooted in different physical underpinnings, intuition from one implementation may enlighten us about the other. DNA is the quintessential information-carrying molecule, and as such it is very interesting to try to find a model of computation that uses nucleic acids as substrates. Attempts at basing computation on DNA as a substrate naturally borrow much from accomplished implementations of electronic computation devices such as logical gates or analog circuits, and in our work we borrow from the concepts of a procedural programming language and a clocked computer, but the beauty of the stochastic and parallel nature of millions of chemical reactions taking place in a tube enriches the model.

As a pragmatic research interest, DNA computation has the potential to become a key component of site-specific therapy, with drugs that can be administered systematically but act only in the correct circumstances *in vivo* [**10**], or detect one of several causes of a symptom and address the specific problem in each case. While we do not yet have a complete implementation ready for human consumption right now, the recent advances in molecular computing, and specifically the ones involving DNA are highly likely to be the bases of such future remedies.

Nimrod Priell

CHAPTER 1

Introduction

1.1. Clocked DNA Computing

We summarize the results of the work by Chang and Shasha on a clocked model of DNA computation [**3**]. They suggest a model of a stored program, based on several "stacks" (complexes of DNA strands) containing sequences of "instructions" (DNA strands with a specified sequence). These instructions detach from the stack sequentially, thereby freeing them to interact with other constituents of the solution they're immersed in, as a result of a reaction with two unique "clock" strands - which act as a fuel driving the computation. The evidence of the advantage of clocked computation and procedural programming surrounds us, as it had a marked effect on the advance of computing from its early, mechanical calculator days. In the context of chemical computation, a stored, clocked program has two major advantages. First, it significantly simplifies the experimental procedure by requiring only the pouring of two clock strands to drive computation, rather than many different kinds of inputs corresponding to the transitions necessary at every step of the computation. Second, it enables the sort of advantages that procedural programming offers in branch predication, iterative structures and function calls, for which synchronous operations are essential.

1.1.1. The DNA. A DNA strand is a linear polymer composed of 4 different types of *bases*, Adenine, Cytosine, Guanine or Thymine, denoted by the first letter of their name. The strand is directional, with one end called the $5'$ end and the other the $3'$ end, so when we write down a sequence, such as $ATTCGG$ we mean the molecule has Adenine at its $5'$ end and Guanine at its $3'$ end. This description of a DNA strand as a sequence of bases is its *primary structure*. DNA strands can combine by anti-parallel complementation where consecutive bases in one strand pair up with consecutive bases in another strand that is oriented with reverse directionality, according to Watson-Crick pairing (A with T, C with G). This is referred to as *hybridization*, and gives DNA molecules their *secondary structure* as they can bend around and pair up with other sections of their own sequence (forming *loops* of various kinds), or they can pair with another, separate strand and form *DNA complexes*. The clocked DNA stored program model extensively uses these complexes, but does not use loops at any point. A specific sequence appearing in a strand is called a *domain*. We can then abstract away from the specific sequence and simply say a certain strand includes the domains a followed by b, whatever the sequences a and b stand for.

DNA is ideally suited for chemical computations for many reasons: It is very stable under a wide range of conditions (in terms of temperature and pressure) with which running experiments is comfortable and easy, it is relatively cheap to obtain in any form, sequence or quantity, its kinetics and thermodynamics have been

researched and modeled accurately for well over 20 years, and most importantly, its sequence-based nature allows us to achieve a wide range of kinetics and specified interactions that are required to ensure the computation progresses as intended with very high probability.

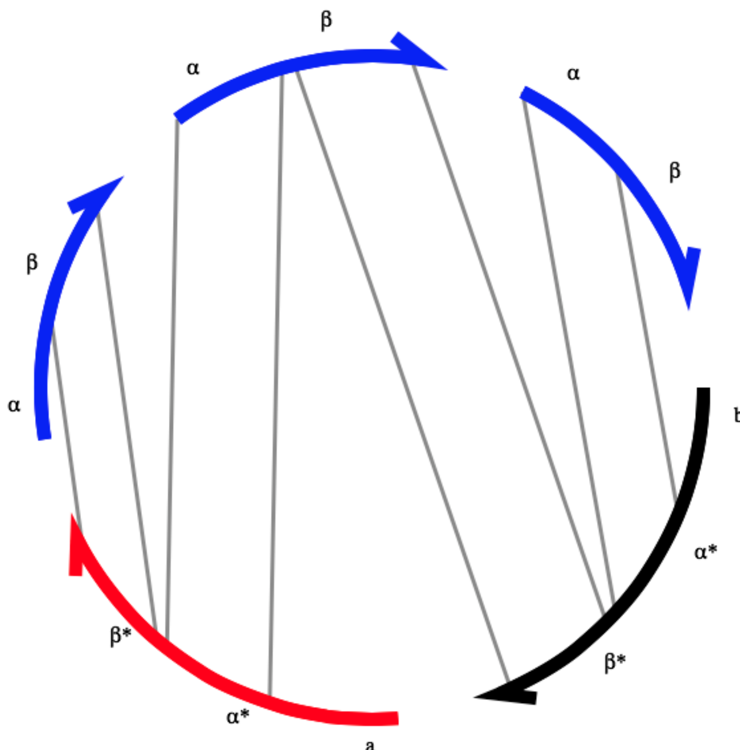


FIGURE 1.1. A two-instruction stack, containing instructions a followed by b, in an unpseudoknotted polymer graph notation[**6**]. α and β constitute the "glue" strand that binds instructions together, and a star denotes complementation. In solution, the complex is linear, and has a protruding α region ("sticky end") to which a 'tick' instruction can bind.

In our model, an *instruction*, which we can represent by a label (e.g out of $a, b, c, \ldots$), is a single-stranded DNA composed of a sequence of nucleotides followed by a shorter sequence (at the 3' end of the strand), shared by all instructions, which is responsible for binding the instruction as part of an instruction *stack*. A stack is a DNA complex, consisting of several instructions, bound together using short strands which bind to the sequences at the end of instructions, as in Figure 1.1.

A procedure for creating these stacks efficiently has been developed and described by Aidan Daly in [**3**]. Once made, a stack can be "executed", or unwound, by pouring in 'tick' and 'tock' strands, which complement the stack binding strands described above. The sequences for 'tick' and 'tock' are generated in a way that ensures they will not bind to any other instruction, and provide the desired kinetics. This is done automatically in the compiler we describe in the following chapters, using NUPACK software [**26**].

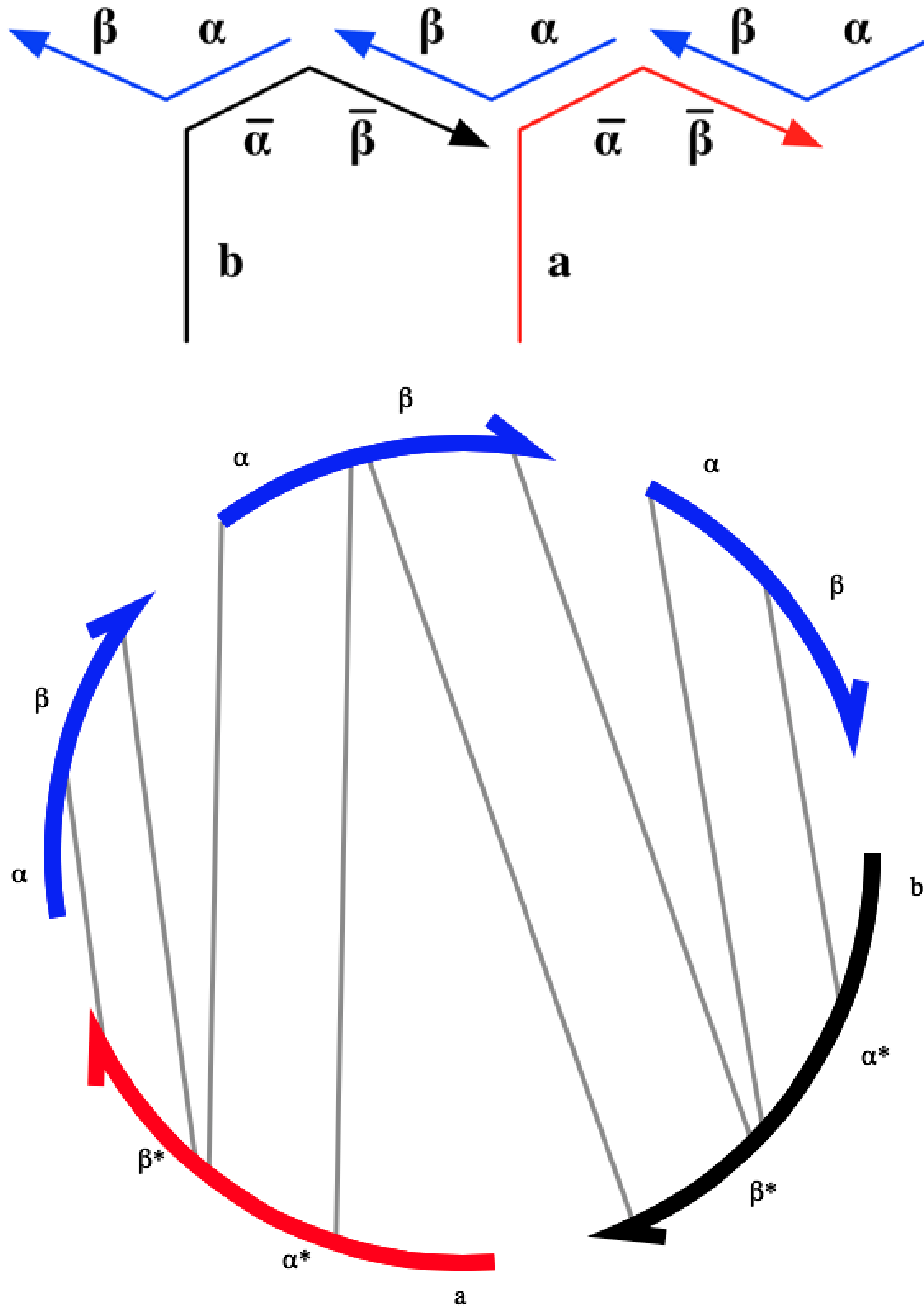


FIGURE 1.2A. The stack complex as in Figure 1.1, and the equivalent linear diagram, representing the binding of stacks as it is schematically in the solution. The cycle diagrams are generated automatically for arbitrary stacks by the code we provide as part of our clocked DNA compiler. Complementation in cycle diagrams is denoted with a star here, i.e $\bar{\alpha} = \alpha*$.

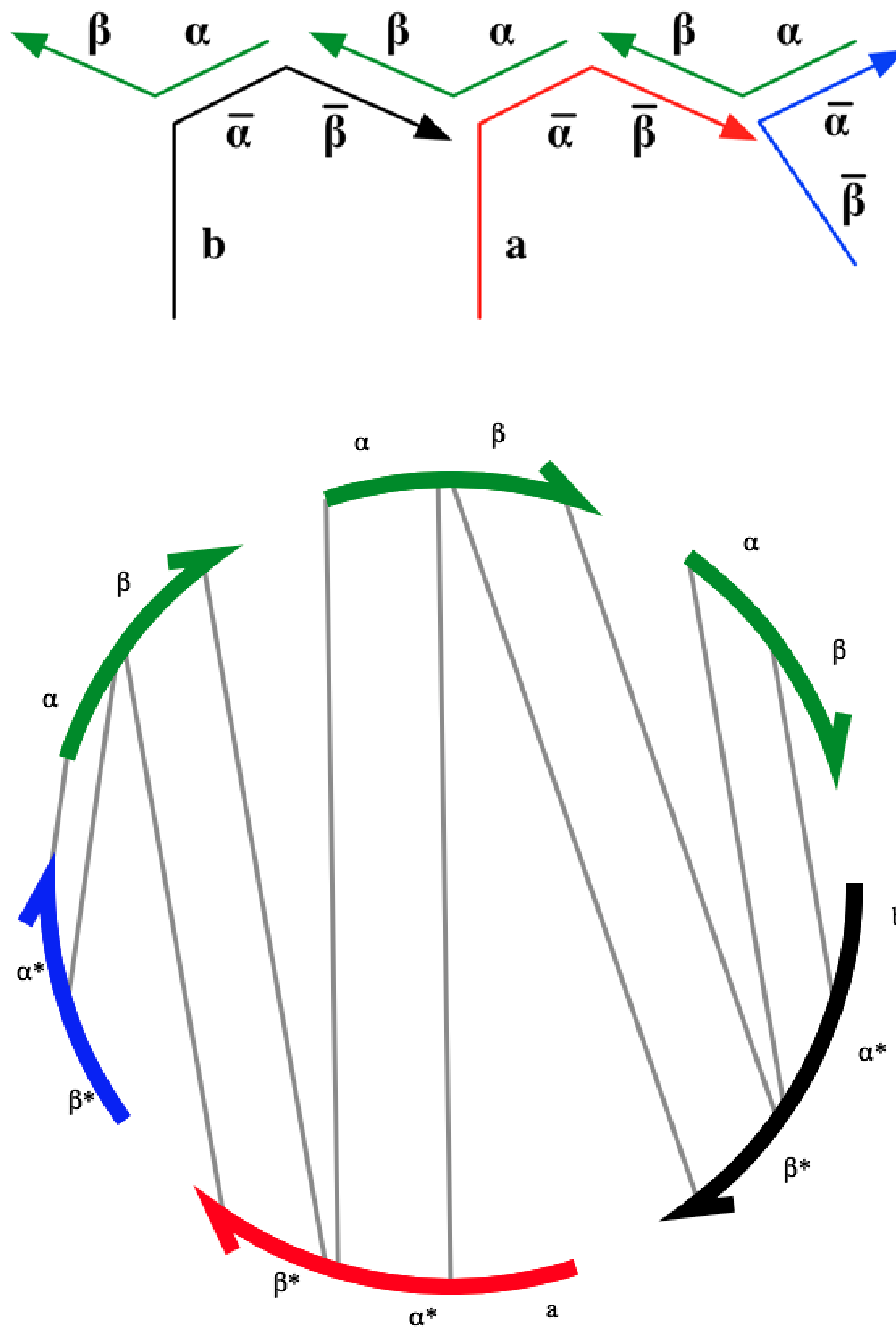


FIGURE 1.2B. After adding a 'tick' strand (blue here), it binds to the free α toehold at the start of a stack (green).

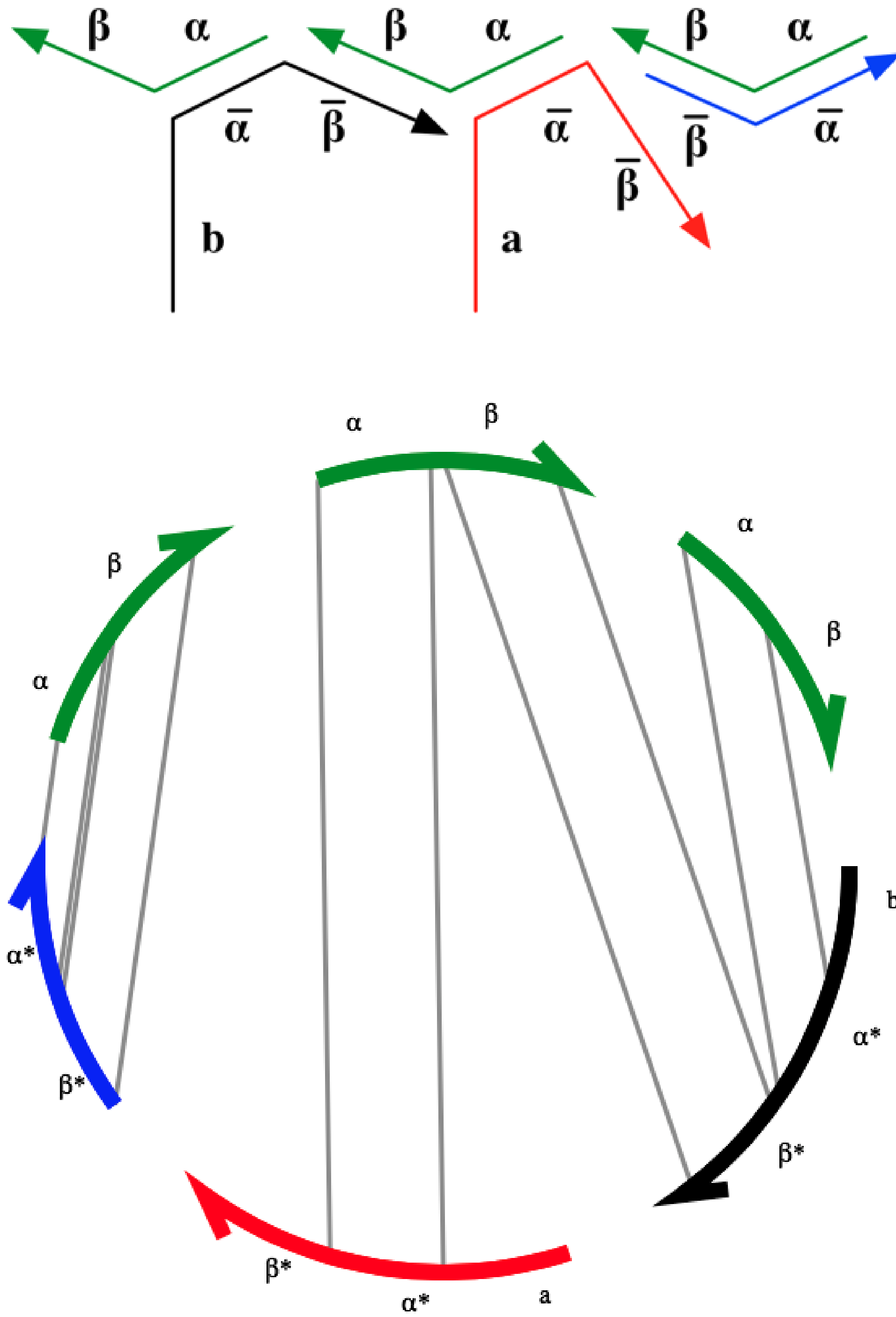


FIGURE 1.2C. Utilizing strand displacement, the 'tick' strand (blue) binds preferentially to the 'anti-tick' binding region (green).

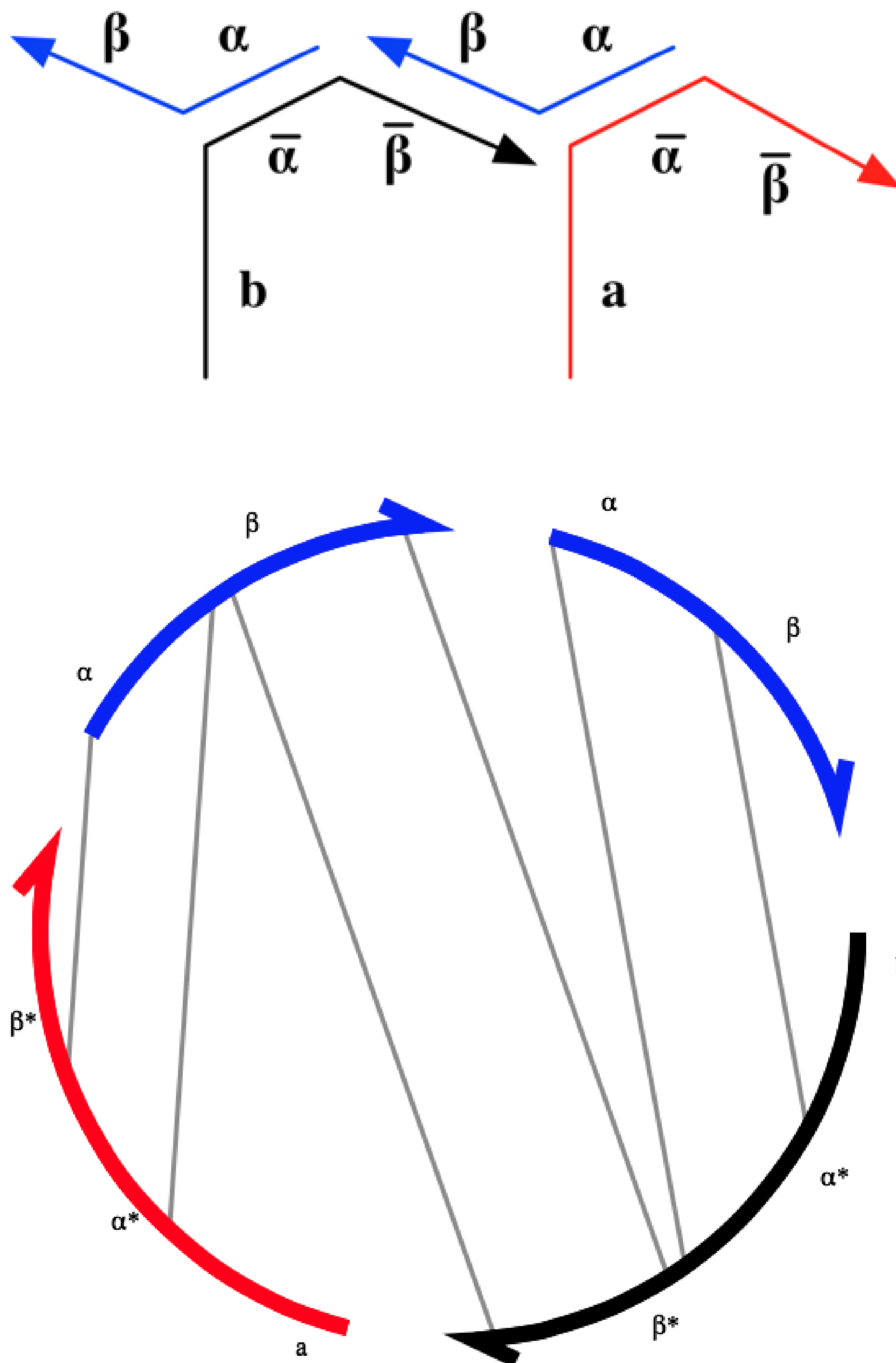


FIGURE 1.2D. The completely bound 'tick'-'anti-tick' complex is removed from the diagram as it can no longer further react. The β-complementary region of the first instruction in the stack, a (red), is free to bind to an incoming 'tock' strand.

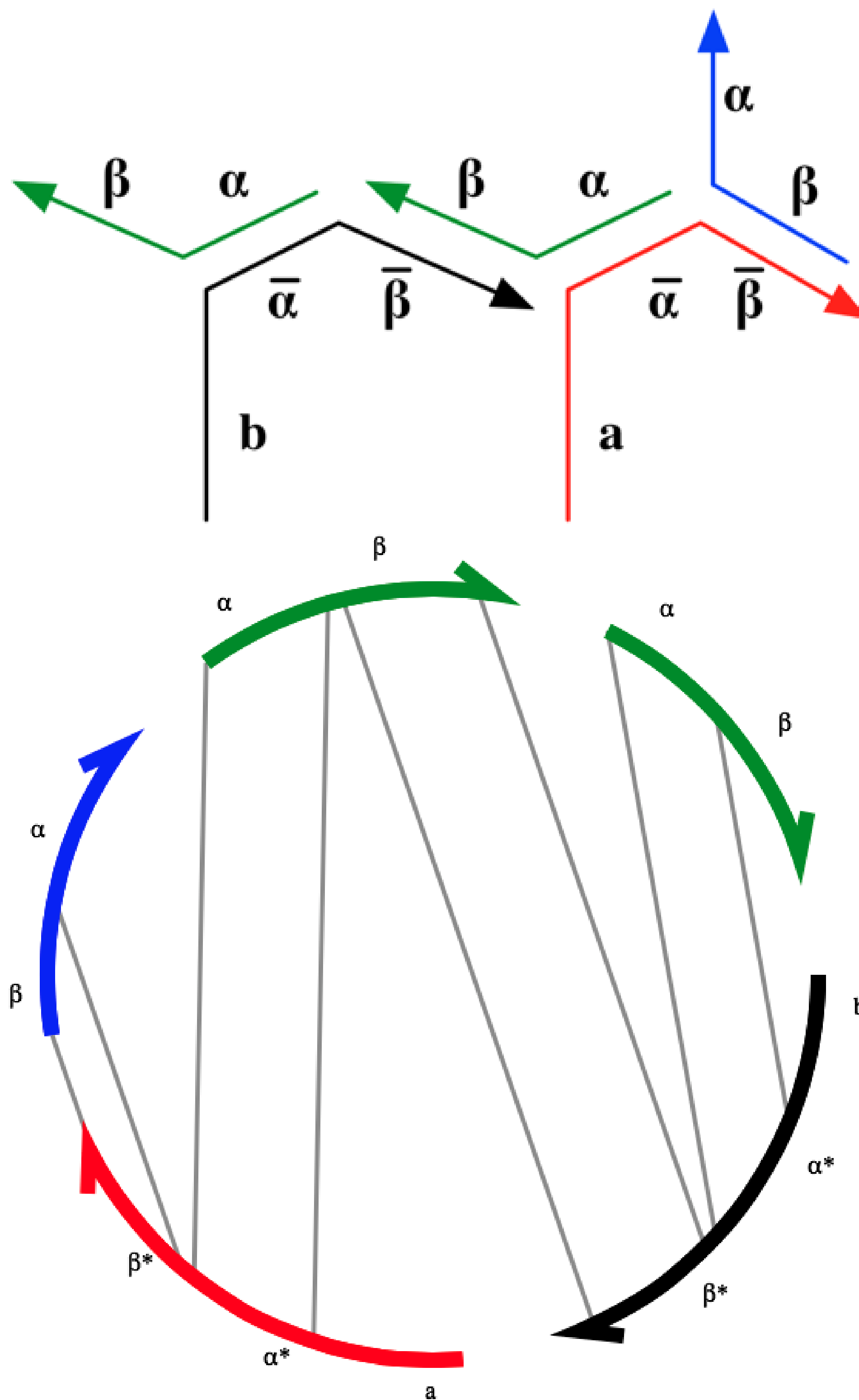


FIGURE 1.2E. The ‘tock’ strand (blue) hybridizes partially with the $\beta*$ toehold on the instruction a (red).

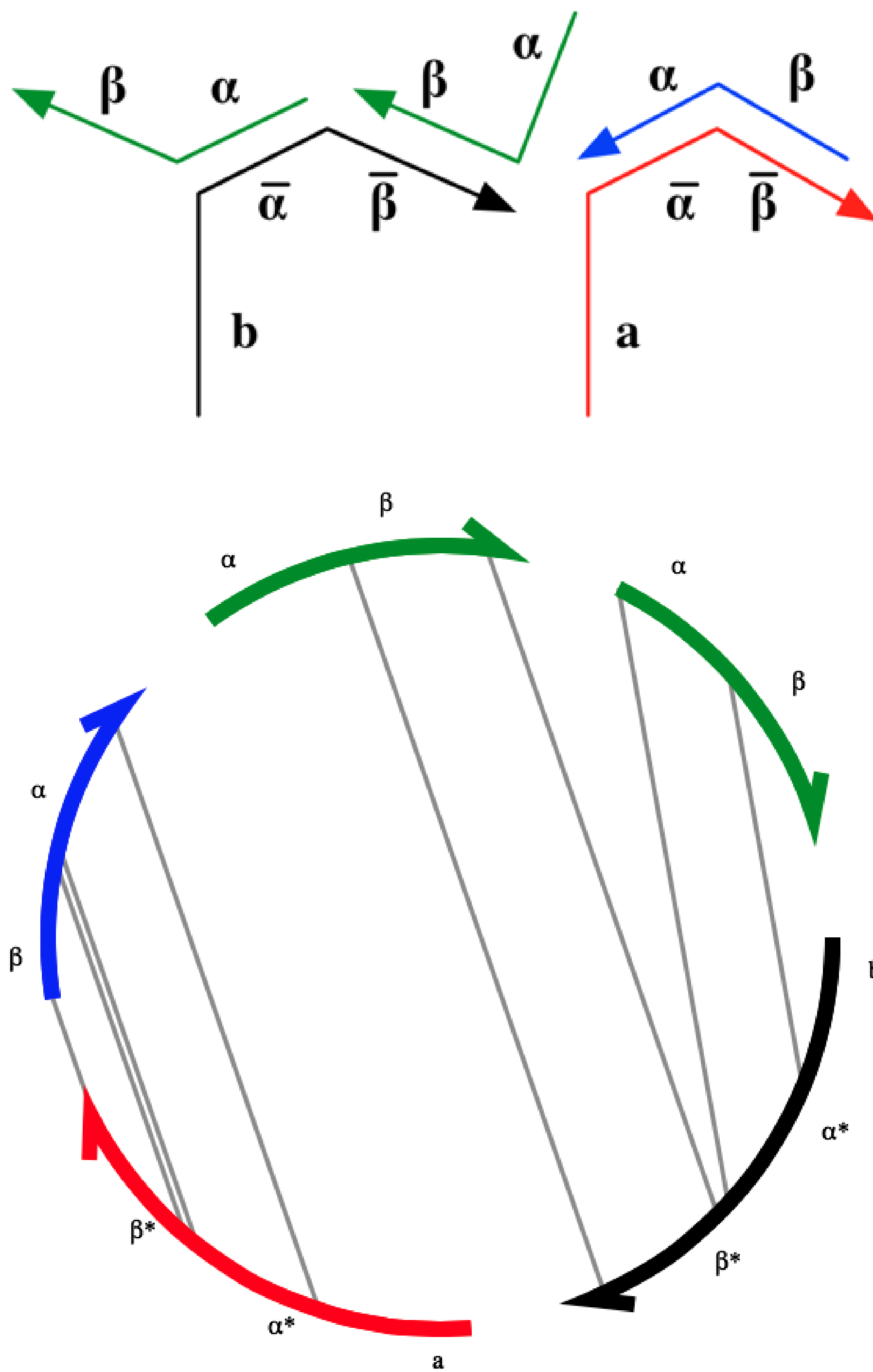


FIGURE 1.2F. Utilizing the same strand displacement mechanisms, the 'tock' strand (blue) binds fully to the instruction (red), leaving the stack one instruction shorter but at the same state as (1), and the instruction a free of the stack, available for reaction with other constituents of the solution.

1.2. Gated stacks

A solution may contain large quantities of stack complexes, and we will be able to pour in 'tick' and 'tock' strands (henceforth referred to as *clock* strands) at quantities only approximately equal to the number of stacks. Because of that and because of the underlying stochastic nature of the chemical process, we cannot assume exactly one instruction will be released off the top of each stack every time we pour in clock strands. Hence, every time a stack contains more than a single instruction, and clock strands are added to the mixture, we must assume some stacks will very likely unwind a little or not at all while others may unwind all the way. These stacks which release more than one instruction are known as *runaway stacks*. They will not constitute the majority of stacks, because the rates of reaction are such that operating on available instructions at the tops of stacks is favorable to unwinding the same stack twice. Nevertheless, runaway stacks will certainly occur. The rates of reaction and resulting distribution of stack unwinding states is discussed further in chapter 3. However, Chang and Shasha also describe a stack *gating* mechanism which can be used for synchronizing the execution so that instructions are not released before we intend them to [**3**].

A *gated stack* contains one or several additional strands, preceding its instructions, bound to the stack in a way that prevents the clock strands from unwinding it. The only way to unwind a gated stack is by first removing the gates using their complementary sequences. The chemical reaction is a strand displacement reaction, similar to the one used for unwinding the stack. In this manner, one gate may be removed at a time. The gate strands may be designed to complement any specified sequence we desire, and are oriented so that they may hybridize with free instructions in the solution. Thus we establish a means of ensuring a stack will unwind only once its gates are removed (gates must be removed in sequence), and thereby allow stack unwinding to occur uninhibited only for ungated stacks. By combining several different stacks (i.e containing different gates and different instructions) in a mixture, and using the reaction kinetics of strand displacement, we can ensure the process proceeds in the intended order with very high probability [**27**]. Hence, we have a general mechanism for synchronous and asynchronous computation taking place solely using DNA.

Furthermore, this system is capable of storing programs and executing them by the simple addition of the same DNA strands into a mixture repetitively, and can be efficiently simulated to ensure that proper results will be achieved *in vitro* as they do *in silica*. In essence, this gives us a simple model with which to think about the sort of constructs found in procedural programming languages on a computer and recreate them in DNA.

In the following chapters we discuss several such programs in length. However, to give the reader an example to motivate the discussion, we consider the following problem: Suppose we have a fluorescent red marker (denoted R), and a fluorescent green marker (denoted G), which we can detect in the solution. Once activated, they flash momentarily, and consequently turn off. Suppose also that there is a single-stranded DNA segment in the solution, perhaps from a bacteria or virus, and we wish to detect if it is one of 4 possible types: a, b, c or d. This is a simple variation of a classic logic riddle. One possible solution could be summarized in the following logic (see Figure 1.3 below.)

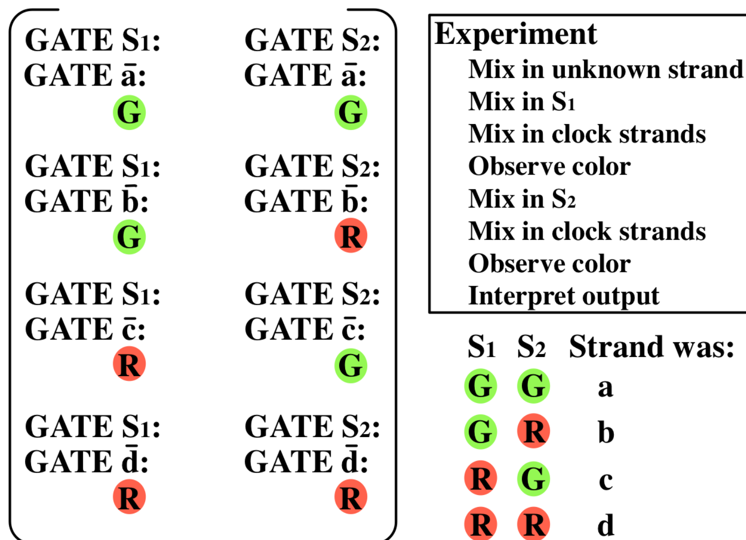


FIGURE 1.3. A simplified gated stack model for differentiating between 4 domains. The constituents of the solution are denoted in brackets. The experiment progresses by following the steps in the top-right square. The table at the bottom-right section indicates the output: Seeing two red flashes, for instance, corresponds to the unknown strand having the sequence d.

In a first set of stacks, we have 4 stacks. Two stacks, one gated by the $\bar{a}$ domain complementary to the a strand, and one by the $\bar{b}$ complementary to the b strand, release the green fluorescent molecule. The two other stacks, gated by $\bar{c}$ and $\bar{d}$ respectively, release the red fluorescent molecule. In a second set of stacks, $\bar{a}$ and $\bar{c}$ each gate a stack releasing the green fluorescent molecule, while the $\bar{b}$ and $\bar{d}$ domains gate a stack with the red fluorescent molecule.

We now have a coding scheme by which, we can attach a sequence of lights to the domain that had to be in the mixture to trigger it. If we see a green light followed by a red light, for instance, we know the sequence labeled a must have been the one in the solution.

The importance of the gates in this example is twofold. First, they have a branching function: We receive different outputs for different inputs because the gates only allow the correct stacks to unwind. Second, it synchronizes the steps of computation: We can ensure that only the first set of stacks is free to react during the first stage, and the second set of stacks will only come into play after we had the chance to measure the effect of the first set in isolation.

1.3. Existing work

We compare our model to outstanding work done with chemical computation, specifically using DNA as a substrate, in the past two decades. A general form of computation with DNA is afforded by programming finite automata using cleavage and ligation proteins[**2**]. Solutions to classes of problems such as boolean logic

3-SAT [**15**] or directed hamiltonian paths [**1**] have been worked out. An electrical-engineering influenced framework for creating both logic circuit and analog circuit equivalents with DNA had also seen significant developments [**7**] [**25**]. Using restriction enzymes, DNA programs capable of answering some queries by propositional logic deductions have been proposed [**19**]. Controlled self-replication leading to computation by a model like cell automata [**17**] and several general frameworks for Turing-complete computation by utilizing chemical reactions with differing rates have been described [**12**] [**14**] [**24**]. A primary inspiration for our work here has been the general computability (Turing completeness) of Stochastic Chemical Reaction Networks (SCRNs) proved by Winfree,[**24**]. His work demonstrates how DNA strands of specific lengths can serve as the chemical compounds demanded by these SCRNs, using toehold-mediated strand displacement kinetics as the reactions taking place in the network [**20**].

Compared to the work reviewed above, our model possesses two important properties. First, it can be applied to a wide range of problems solvable by a finite stochastic automaton. Second, it is very specific in that it prescribes the exact nucleotide sequences and required complexes as well as the progression of the computation and expected yield, using our compiler. It also enjoys an important practical advantage: Computation is based solely on DNA molecules, which are relatively cheap and easy to obtain, are stable for long durations of time under standard conditions and can be used in conditions mimicking a living cell.

CHAPTER 2

Chemical underpinnings

2.1. Introduction

Computation can be viewed as a discrete process – a sequence of steps – in which initial data is being transformed according to some rules, until it reaches an end state, the result of the calculation. These transformations occur either autonomously or as a result of some external stimuli (input). Upon completion, the output must be extracted by some means. In this chapter we will review each of these components in the context of our model.

In our model, states of the computation are described by the molecules contained in a dilute solution under constant temperature and pressure conditions. The number of molecules used is assumed to be sufficiently large so that the thermodynamic limit equilibrium is reached and small-population effects can be neglected. Each state transition corresponds to pouring more DNA strands into the mixture: Either clock strands or input molecules containing one or several labeled domains. We then need to consider both the kinetics of the transition and its thermodynamics. The kinetics govern how new species are generated as a result of interactions between the molecules already in the solution and the input molecules, while the thermodynamics govern the new equilibrium reached.

2.2. Thermodynamics

The thermodynamics are derived from knowing the free energies of the different possible complexes and the partition function. The change in free energy between two single stranded DNA molecules and their hybridized, double-stranded DNA complex has been found empirically to be accurately modeled by the nearest-neighbour formula [**18**],

$$\Delta G^\circ_{\text{total}} = \sum_{i=1}^{n} \Delta G^\circ(p_i) + \Delta G^\circ_{\text{init}}(p_1) \tag{2.1}$$

Where $p_1, \ldots, p_n$ are *pairs* of neighhboring bases on each strand (e.g AG pairing with CT) and $\Delta G^\circ_{\text{init}}$ is an extra term affected only by the first, or initiating, base pair (which is either A pairing with T or C pairing with G). The key here is that there are no long-distance effects or length-dependent effects on the change in free energy: All that matters are the pairing bases and their immediate downstream neighbors. Values for the enthalpy change ΔH° and entropy change ΔS° for the ten possible nearest neighbor pairs and the initiation factor have been empirically found for $37^\circ C$ and are presented in SantaLucia's paper[**18**]. $\Delta G^\circ(p_i)$ can be derived using $\Delta G^\circ = \Delta H^\circ - T\Delta S^\circ$ and extrapolated to other temperatures in a reasonable range where we do not expect the heat capacity C_p of the strands to change much. Another term can treat solutions in which the salt concentration is

other than 1M NaCl, but we refer the interested reader to the original paper [**18**] for further details, whereas we just outline the use of these formulas in our model.

We can now use these Gibbs free energies to calculate the *partition function*

$$Q = \sum_{\Omega} e^{-\Delta G(s)/kT}$$

where k is the Bolzmann constant, T is the temperature and we sum over all possible complex configurations (base pairings) in a complex containing the specified strands. This can be used to calculate several desired quantities for our simulation of the computation, such as the probability of a specific configuration, given by

$$P(\Omega) = \frac{e^{-\Delta G(\Omega)/kT}}{Q}$$

This can be further used to design sequences that are highly likely to bind to each other in some predetermined way.

To actually compute the partition function and design the sequences that are used for the various instructions, clock strands and gates we rely on the NUPACK nucleic acid design and analysis software. NUPACK includes algorithms that comprehensively and precisely model interactions among several DNA or RNA molecules. However, NUPACK requires that the molecules in a complex be *unpseudoknotted*.

2.2.1. Pseudoknotted Complexes. A complex of DNA strands can be represented by first ordering the involved DNA strands in some way, and then enumerating the bases of all comprising strands in 5'-to-3' fashion sequentially. For example, a complex comprising of the two strands 5'-ACCG-3' and 5'-TTAC-3' can be enumerated in the order presented, so the index of the first A will be 1, the index of the first T will be 5 and the index of the terminal C will be 8. We can then define the set P of paired bases (i, j) in the complex, where a base pair is always written with $i < j$. An unpseudoknotted complex is one that satisfies a *nesting* property: $\forall\ i, j, k, l$ s.t. $(i, j), (k, l) \in P,\ i < k < l < j$. A *polymer graph* is a representation of the complex with the strands drawn as arrows, pointing to the 3' end, in a circle. Edges between the arrows represent paired bases. The reader may refer to Figures 1.1 and 1.2A-F for examples. An ordering of the strands is derived from this representation.

A complex $\Omega = \{s_1, s_2, \ldots, s_k\}$ is *connected* if it cannot be written as $\Omega = \Omega_1 \cup \Omega_2$ where $\Omega_1 \cap \Omega_2 = \emptyset$, and there are no $(i, j) \in P$ with $\{i, j\} \cap \Omega_1 \neq \emptyset$ and $\{i, j\} \cap \Omega_2 \neq \emptyset$. When this property fails to hold, i.e in a *disconnected* complex, it means there are Ω_1 and Ω_2 such that there are no pairs connecting both complexes, and they can travel separately in the solution. Any unpseudoknotted complex can be drawn in a polymer graph with no intersecting edges, and if it is also connected, there is a unique way of doing so, up to rotations. That is the content of Theorem 2.1 (Representation) in Winfree et al. [**6**]:

THEOREM 2.1. *For every unpseudoknotted connected secondary structure* $\Omega = \{s_1, \ldots, s_k\}$*, there is exactly one circular permutation* π *on* $\{1, \ldots, k\}$ *that yields a polymer graph with no crossing lines.*

We further note that for complexes without the nesting property, any ordering of the strands will yield a graph with intersecting edges.

It is key, then, that we demonstrate that at every step, all complexes involved are unpseudoknotted. Otherwise, we cannot justify using NUPACK's algorithms as an accurate calculation of the partition function.

All multi-DNA complexes in our computation fall into these two categories: They consist of either a stack, or a complex resulting from interactions of instructions with other instructions or input strands. Stacks may be gated, and they may be in some stage of unwinding, as portrayed in Figures 1.2A-F.

We consider these interactions to be the only ones possible because our compiler ensures that all domains comprising either inputs, gates or instructions have sequences that are distinct from the ones used for clock strands. Furthermore, the process of stack unwinding is designed so that an instruction is never exposed at the end of a stack. The physical conformation of the stack complex inhibits any polymer whose length is comparable to the length of an instruction from binding to the single-stranded "free-hanging" instructions in a stack. As no polymers of shorter lengths are considered by the compiler in the first place, asides for clock strands which will not interact with instructions due to their sequences, these sequences run no risk of creating unwanted reactions. An example of this principle in action can be portrayed by the following plot, which corresponds to the stack in Figure 1.2a. In this plot we can see how the nucleotide sequences chosen for the clock strands and instruction domains create extremely stable stacks. The only probable configuration of the complex is the one we had designed – i.e whose only hybridized regions are these that bind the stack's instructions together. These probabilities are based solely on the free energies of the nucleotide sequences and do not include any steric effects.

Taken together, these considerations mean that we can assume stacks interact only with tick and tock strands, and instructions interact only with other instructions, gate domains of gated stacks, or input strands. Additionally, our model's semantics do not allow complicated binding patterns for instructions or input strands that may result in pseudoknotted complexes, single-stranded loops or multi-strand constructs: These arise from partial interactions between sequences, whereas our compiler ensures that every domain is assigned a unique sequence minimizing unwanted interactions. The only interaction with significant probability of occurring is the complete hybridization of a domain with its complement. This always creates a double-stranded complex that is easily seen to be unpseudoknotted: It contains only two strands bound over a consecutive sequence of pairs in anti-parallel fashion. Reading the strands in the ordinary $5' \to 3'$ direction, the binding starts from base i_1 on one strand, extending to base j_1 on that strand, and from base i_2 through to base j_2 on the other strand. Without loss of generality, we can start the enumeration so that i_1 is the smallest of the numbers (that is, we may pick for i_1 any of the two strands). Since i_1 and j_1 are on one strand, and i_2 and j_2 are on the other, we have $i_1 < j_1 < i_2 < j_2$. Since DNA strands hybridize in opposite directions, i_1 binds to j_2, $i_1 + 1$ to $j_2 - 1$ and so forth until (j_1, i_2) form a base pair. This is easily seen to satisfy the nesting property.

In fact, the same principles hold in a stack is throughout its unwinding. This can be asserted by examining Figure 1.2a which we recommend the reader consult with to clarify the argument: We choose the following ordering for a stack $| \, a_1 a_2 \ldots a_m$: First its instructions in reverse order $(a_m, a_{m-1}, \ldots, a_1)$, then $m + 1$ binding strands consisting of $\alpha - \beta$ domains (in $5' \to 3'$ direction). Each instruction

consists of its domain (a_i) followed by $\bar{\alpha} - \bar{\beta}$ domains. In terms of domains, this ordering gives us:

$$a_m, \bar{\alpha}, \bar{\beta}, a_{m-1}, \bar{\alpha}, \bar{\beta}, \ldots, a_1, \bar{\alpha}, \bar{\beta}, \alpha, \beta, \alpha, \beta, \ldots, \alpha, \beta$$

In Figure 1.2a this can be witnessed by starting from the instruction b (black) and proceeding in a clockwise fashion.

The binding of the $\bar{\alpha}$ domain in the i'th instruction with the α domain in the $i+1$'th binding strand and of the $\bar{\beta}$ domain in the i'th instruction with the β domain in the i'th binding strand completely describes a correct construction of the stack. This binding pattern includes only nested pairs: The ordering we chose for the strands means the first $\bar{\alpha}$ domain encountered hybridizes with the last α domain, the following $\bar{\beta}$ domain binds to the one-before-last β, which precedes the last α domain. The following $\bar{\alpha}$ domain then binds to the α domain preceding the second-to-last β domain, and so forth. The anti-parallel binding of DNA means that inside each domain the nesting property is maintained. This argument is much better elucidated by looking at the corresponding figure and witnessing no lines cross each other. It continues to hold true for the rest of the transformations we make to the complex as we unwind an instruction from it, as well as for domains that unlock the gates and the stacks they interact with. We do not bother the reader with details as they require only minor modifications to the general scheme above. It can be simply stated that all of these transformations on the basic stack structure involve operations at the deepest nesting level in the list of pairs, and none of them violate the nesting property. Figures 1.2B-E are examples of how binding is altered at the deepest level of nesting when clock strands are introduced to unwind a stack.

2.3. Kinetics

The kinetic model we use is that of DNA hybridization and toehold-mediated strand displacement. These reactions can be accurately controlled so as to virtually eliminate unwanted reactions, and control the rates of reaction and resulting equilibrium distributions [**27**]. This control is exerted by varying the sequences and the length of the toehold used. A *toehold* is a single stranded section at the end of a hybridized pair of DNA molecules. It is present whenever the ends of the two composing strands do not have the exact same sequence, or one is shorter than the other, as opposed to *blunt* ends which arise whenever there is an exact match between the ends of the strands. Our stack design extensively exploits toeholds: In all steps in the unwinding of a stack there is a toehold present at the end of the stack to catalyze the reaction leading to the next step in the unwinding.

A toehold-mediated strand displacement is a reaction in which 3 single-stranded DNA molecules are involved. Let us refer to them as A, B and C. The reaction begins with A hybridized to B, and ends with A free in the solution, and B and C hybridized: Effectively, C has displaced A from its bound state with B. This can obviously occur only if A and C share a sequence which is complementary to some region in B. However, If the region where A complements B is the exact same region where C complements B, this reaction will be exceedingly slow: In order for C to displace A, A must first completely disassociate from B - a highly energetically unfavorable reaction, and then C must be near B so that they may hybridize. However, if there is a way for C to be near the region of B that it competes on with A, the reaction may actually be likely. This can happen, obviously, when C

is already bound to B. Specifically, if the relevant region of A extends all the way to one end of A's sequence, and C and B share a region of complimentarity that includes the region A and B share, but also extends beyond that region, where A has ended, then C can partially bind to B while A is still bound. This co-locates C with B, so that random disassociations of the end of A from B are competitively replaced by C binding to B. In this manner, A unbinds from B while C takes over until A detaches completely – bringing us to the state where A is free while B and C are bound in both the region where A had previously been and the extending sequence which assisted in the displacement of A by C. This region of extension is B's toehold for C, and the effect is especially accentuated if the toehold extends all the way to one end of B's sequence.

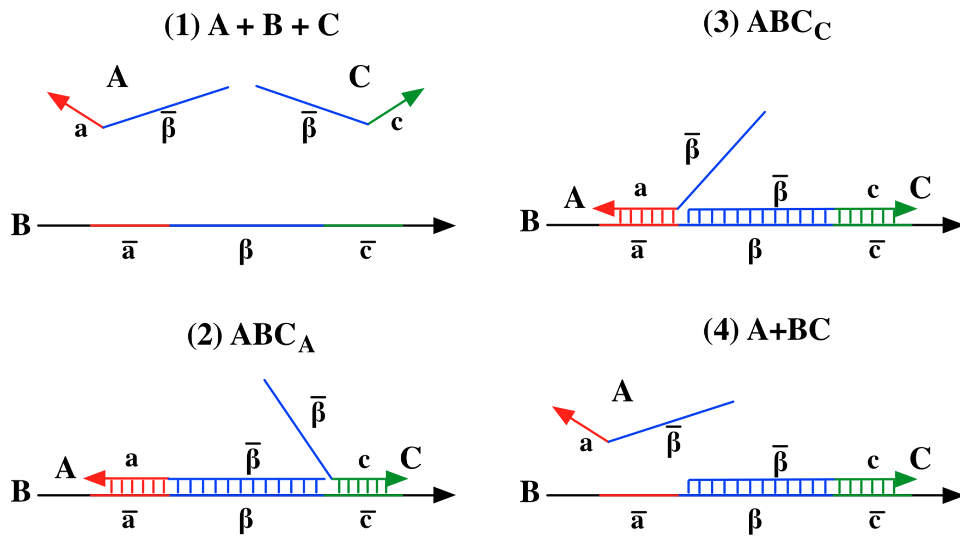
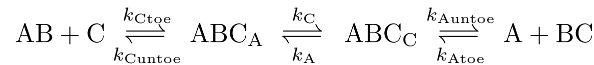


FIGURE 2.2. Stages of toehold exchange. Strands A and C share a domain $\bar{\beta}$ complementing B's center domain β, but each also complements B at a different region flanking β. In (2) C collocates with B allowing it to initiate strand displacement and replace A's binding of the β region. In (3) this step is complete, and toehold-mediated strand displacement has occurred. Here $\bar{a}$ is a short toehold for A. The process may reverse or it may proceed. If it proceeds, we are in (4), where A disengages from B and a toehold exchange had occurred.

The process, depicted in Figure 2.2, can then be described by the following 3 steps:



$$\mathrm{AB} + \mathrm{C} \underset{k_{\mathrm{Cuntoe}}}{\overset{k_{\mathrm{Ctoe}}}{\rightleftharpoons}} \mathrm{ABC_A} \underset{k_{\mathrm{A}}}{\overset{k_{\mathrm{C}}}{\rightleftharpoons}} \mathrm{ABC_C} \underset{k_{\mathrm{Atoe}}}{\overset{k_{\mathrm{Auntoe}}}{\rightleftharpoons}} \mathrm{A} + \mathrm{BC}$$

Where ABC_A denotes C hybridized to the toehold on B and A completely hybridized to B, and ABC_C denotes C hybridized completely to B and A only hybridized to B at regions extending beyond the regions where A and C overlap, if any such regions exist. The different rates of reaction correspond to C hybridizing or de-hybridizing with the toehold region (k_{Ctoe} and k_{Cuntoe} respectively), exchange of places in the overlapping region (k_C for C displacing A, and k_A for A displacing

C), and finally, A disassociating from B in any remaining region of complementarity. The latter can symmetrically be viewed as B's toehold for A, and is hence denoted k_{Atoe} and k_{Auntoe}. The toehold binding and unbinding rates of reaction are derived from the very same DNA hybridization free energies discussed above. Values for k_C and k_A are the result of modeling the branch migration between A, B and C as a symmetric random walk with absorbing barriers, on the sequence of B [**23**]. The expected time for the walk from the state ABC_A to ABC_C can be calculated and k_C effectively becomes its reciprocal: $\frac{1}{k_C}$ is proportional to the first passage time of the walk to the end of the overlapping complementarity region, which is the length of that region squared. All intermediate states in the walk are transient, so that in the limit we find that the molecules are in a dynamic equilibrium between the two end states. The probability for each state is determined by the free energies of both complexes. It is acknowledged in the literature that this model is somewhat simplistic in its neglect of sequence-specific step rates and its treatments of the initiating step as a barrier rather than allowing a cancellation of the random walk. Nevertheless the model is accepted as an approximation and agrees with experimental data to a large extent.

The three-step process above is a slight generalization of toehold-mediated strand displacement in that it allows A to have a region of complementarity with B which overlaps but is not fully contained in C's complementary region. If A's complementary region is indeed contained entirely within C's, we may drop the intermediary ABC_C step, effectively treating k_{Auntoe} as immediate (or rewriting the rate equation with $ABC_C = A + BC$). The full three-step process is termed a *toehold exchange* rather than strand displacement. This is because it begins with A and B hybridized and a toehold on B available for C to bind to, and ends with a different toehold, the one for A, available for binding. The largest contributions to the rate of the entire reaction are made by the lengths and nucleotides chosen for A and C's toehold domains, and by varying the toehold length anywhere between two and eight nuncleotides long we achieve a degree of freedom in the forward and backward reaction rates for $AB + C \leftrightarrow A + BC$ that ranges over six orders of magnitude. This is the mechanism, alluded to in Chapter 1, allowing us to guarantee reactions take place with high probability, specificity and in the correct order.

2.3.1. Simulating toehold exchange processes. While toehold exchanges underlie much of the mechanics of our computational model, they are not included in the simulations our compiler provides for DNA programs. There is a stochastic simulator made specifically for toehold exchange driven processes, called VisualDSD[**11**]. Unfortunately the source code for the project is proprietary and unavailable, making it somewhat difficult to combine its capabilities with the requirements of our compiler. An interesting future project may be to leverage its algorithms and significant power in stochastic simulations as a part of the output our compiler provides, as an addition to the thermodynamic data we produce.

2.3.2. Limitations. We note that several assumptions are made here that might not accurately reflect the realities of an actual experiment. For one thing, there are several ΔG° contributions that are not considered by the papers we consulted. An example may be contributions from the secondary structure of the stack that may behave differently than the model due to the existence of “hanging” single

strands, also referred to as *coaxial stack dangles* [**8**]. Furthermore, there is intentional neglect of reactions considered to be so unfavorable or so slow that they result in very little products during the course of computation. This is done to avoid an exponential increase in the size of the resulting state space. Such is the case when we consider all domains to be either entirely hybridized or completely un-hybridized. In an actual solution we may find a dynamic equilibrium with partially hybridized domains very rapidly associating and disassociating a single base at the ends of the domain, for example. These omissions are standard approximations and considered safe as they are generally second- and higher-order effects and will not significantly alter the result of a computation where order of magnitude differences in output are used to differentiate the terminal states.

Different kinds of limitations arise from the conditions under which the model is applicable. One is that we consider a dilute solution, specifically under a critical concentration of molecules demanded by the kinetic model of toehold-mediated strand displacement [**27**]. Another stems from the accuracy of the thermodynamic parameters calculated by NUPACK software and used by our compiler to predict the efficiency of the calculation. The accuracy of these parameters holds only in specific ranges of specific salt concentrations, temperatures and so forth. These are fairly lax restrictions - many experiments in synthetic DNA chemistry including motors, circuits and DNA origami have been performed under these very same conditions. Nevertheless the compiler might require alterations to be useful in a significantly different environment.

FIGURE 2.1. Heat-map plot of the probabilities of a pair of bases (i, j) binding in a stack containing two instructions, at its initial state. The length of the instruction's unique domain is 20 bases and the length of the clock strand is 16 bases. Thus, the total length of an instruction strand is 36 bases and the length of the complete stack including its three binding sequences is 120 bases. The entries in the last column, 121, correspond to the probability of the base i remaining **un**bound, so that it is equal to one minus the probabilities of binding to any other base in the complex. The ordering is done with the instructions first followed by the binding regions, as if we started counting with the first base of the last instruction to be unwound (the one appearing in black in Figure 1.1,) and went around in a clockwise direction on that diagram. In the heat-map, red denotes high probability and white denotes low probability. It is clear that the stack's sequences are such that the correct binding, depicted in Figure 1.1, occurs with very high probability, as the "stairs" match exactly the position of the binding strands hybridizing with the clock domains of the instructions. The bases in lighter red correspond to the first binding element, which has a slightly "frayed" ending.

CHAPTER 3

Probabilistic Automaton Model of Clocked DNA Computing

3.1. Introduction and existing results

The general framework we operate in is one where several species of reactant molecules (DNA strands) and products (hybridized DNA strands) interact in a solution. A fundamental question to ask then, is what kind of computation is theoretically available within that setting? Stochastic Chemical Reaction Networks (SCRN), a model that conceptualizes the interaction of many species in a solution where many possible reactions can take place, have been shown to be Turing universal with just two different reaction rates, via comparison to Petri Nets, Vector Addition Systems and Register Machines [**24**]. That paper contains a discussion of a chemical clock for achieving arbitrary probability of correct state transition at the cost of slower execution, as well as discussion of the computability of Primary Recursive functions. However, the model suggested there is highly theoretical in that it does not specify which species and reactions can be used and how to choose them, rather focusing on the asymptotic bounds to reliability and computational properties. Another paper shows an infinite set of chemical reactions to be Turing complete using a register machine [**12**] and includes a language for programming with these reactions, but again the discussion is at the level of arbitrary molecules. That chemical reactions with various speeds can do basic arithmetic and algebraic operations is portrayed in Riedel et al.[**14**]. There, output is given as the exact number of product molecules so high precision in measurement is required to extract output.

We present a model with more limited computational capabilities that is nevertheless specified in concrete species and reactions. We describe a theoretical framework for computation using our models. We have also developed a compiler that, given an abstract description of the "program", will specify which DNA molecules should be used, under which conditions, and how to expect the computation to progress. Our compiler extends the great work done with NUPACK.

3.2. Clocked DNA computation as an automaton

Taking a higher level view than the one in the preceding chapter, we can look at a molecular computer as a probabilistic finite automaton. This description limits the model's theoretical computational power which, as discussed above, can be shown to be Turing universal. However, many interesting implementations can be realized by probabilistic finite automata and this remains an interesting model for specification and engineering of certain classes of problems.

To model our clocked DNA computer, we introduce the following notation: The program is represented by a graph, whose nodes are states in the computation. Edges indicate transitions between states. Each state describes the contents of the solution that can potentially react. An instruction is identified by a label $(a, b, c, \ldots)$, and a stack of instructions is denoted by a vertical bar $(|)$ containing one or more such labels. $|a$ is the one-instruction stack containing a. The top instruction of the stack is the first one to pop when 'tick' and 'tock' strands hybridize with the stack. Instructions with multiple domains are rarely used, but can be denoted by concatenating the domains delimited by dashes. For example, the instruction a-b-a is a single strand containing the sequences of a, then b, then a again (in the $5' \rightarrow 3'$ direction.) A complement is denoted, as in Chang and Shasha [**3**], by a bar. $\bar{a}$ complements a and they bind to form $a\bar{a}$. Hence we note the complement of a-b is $\overline{a-b} = \bar{b} - \bar{a}$, though, again, this is rarely used. Once two complementary strands pair up, they are no longer free to react: The reaction for melting a long complementary region and hybridizing with a different strand without the assistance of toeholds is so energetically unfavorable under standard temperatures, that we treat these double-stranded DNA molecules as essentially inert. In our state diagrams we will just remove them from the state description, except when they help illustrate a concept.

The edges encode for the inputs of the program (known as *transitions* or *actions* in automaton theory), and they correspond to pouring strands into the solution. We assume the tick and tock strands hybridize solely with their corresponding anti-tick and anti-tock strands on the stack (and we can guarantee this with high probability using our compiler). Thus, we model popping one instruction from the top of any available stack as a single operation instead of bothering with specifying the intermediate steps of the reaction illustrated in Chapters 1 and 2. We denote this action of introducing clock strands by t. Any other species of strands can be poured into the solution. They are denoted as domains in the very same way the instructions are. However, their actual sequence is considered to contain several nucleotides flanking the active domain, so as to prevent unwanted reactions from occurring with strands that are part of the stack. Hence sometimes, a as an instruction and a as a poured-in input might not have the exact same sequence (except at their middle section). Again, our compiler takes care of the underlying sequences, and due to the molecular structure of the stack very few nucleotides on either side of the domain are required. Hence, the difference between the molecules is not important enough to necessitate more notation in the model.

Figure 3.1 contains an example of a very simple state machine following the rules described above. However, it is far from an accurate model for the actual reactions taking place in a mixture and is included just as a basic example of the principles described thus far.

Our real model includes several modifications. First of all, we note unlike a classic automaton (of any variety), which can receive only one input letter per step, our model can receive several inputs at once - for example by pouring in a mixture of $\bar{a}$ and $\bar{b}$ strands. Thus, if Σ denotes the alphabet of possible inputs, our automaton model handles states whose transitions are in 2^{Σ}. An example is portrayed in Figure 3.2.

A natural question to ask at this point is whether the inclusion of multiple-input actions matters, or whether sequentially introducing the same inputs will

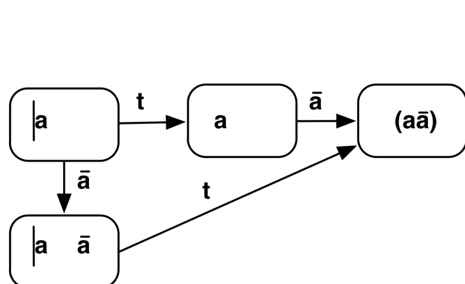


FIGURE 3.1. A simple deterministic state machine in our notation, starting with one single-instruction stack. The final state could be empty, but for clarity we have included the resulting double-stranded complex.

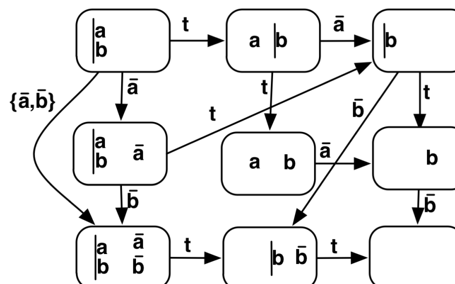


FIGURE 3.2. A deterministic clocked DNA automaton starting with one 2-instruction stack. Not all possible transitions or states are denoted, but enough to give the reader an idea of the possibilities resulting from multiple inputs.

achieve the same result. The answer is that it does matter and in fact even the order of introducing the inputs matters. The reason lies with the nature of the equilibrium that the states are in. Except for toehold exchanges, (which are the mechanism used in order for the tick and tock strands to unwind the stack), it is highly unlikely for a long region of bound duplex strands to unbind and have one of the single stranded DNAs hybridize with a different single-stranded molecule, unless the temperature is very high. For this reason, in a case where we have some a instructions free in the solution, and we introduce first $\bar{a}$ strands and then $\bar{a} - b$ strands, equilibrium is first achieved with the introduced $\bar{a}$ strands, whereby almost all free a instructions form $a\bar{a}$ complexes. The $\bar{a}-b$ strands introduced later remain in the solution as single-stranded polymers. However, if we first introduce $\bar{a} - b$ strands and then $\bar{a}$, all free a instructions will form $a\bar{a} - b$ complexes, leaving none to react with the $\bar{a}$ strands. We then end up in a different state – one where $\bar{a}$ strands, not $\bar{a} - b$ strands remain free to react in the solution. If we introduce them together in a well-mixed solution containing the same number of moles as introducing them sequentially (i.e twice the number of free a instructions), we end up with a state containing both $\bar{a} - b$ and $\bar{a}$ strands free, as approximately half of each population will bind to the instructions.

The discussion above naturally brings us to the other modification we make to the finite automata model - that of the nondeterministic automaton.

3.2.1. Probabilistic nature. The automata described so far can be actually formulated easily as deterministic finite automata, with an appropriate labeling of the states and 2^{Σ} (i.e the Power Set, containing all subsets of Σ) as its input alphabet. However, they are again a very simplistic model of what occurs in the actual solution. In fact, in a mixture, any addition of an input molecule (e.g. a tick and tock strand) will reach some equilibrium between the reactants and the products of all possible reactions caused by the addition of that molecule. This equilibrium distribution is determined by the concentration of the reactants and products, and by the rate constants of the reaction. In order to describe this equilibrium, we treat each state as describing a single product configuration of the several possible ones achieved. Multiple states are reached by each input

transition. We do neglect the states that are thermodynamically unstable in the sense that we do no expect them to exist in equilibrium - for example hybridization between domains with non-complementary sequences. In Chapter 2 we discussed the underlying chemistry necessary to simulate this process, and addressed the actual computation of the distribution among these resulting states. Here we satisfy ourselves with ascribing some variables to these probabilities.

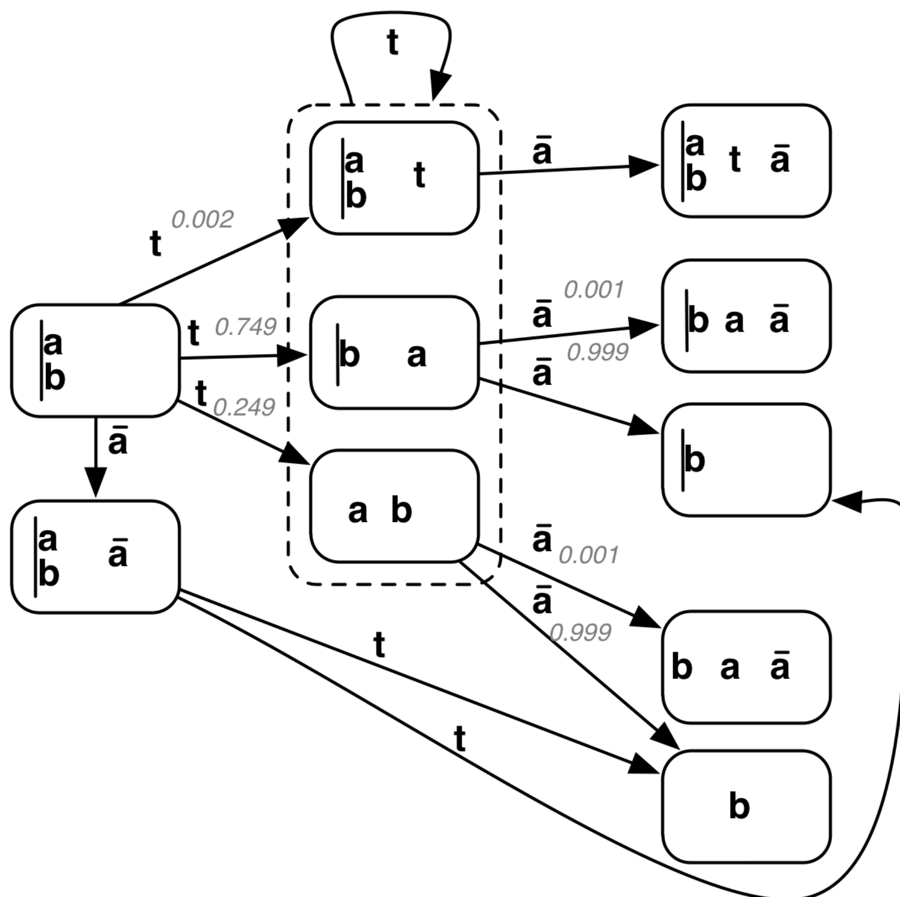


FIGURE 3.3. An example of a partial probabilistic finite state automaton starting with a single two-instruction stack. Not all transitions are marked. Example probability distributions are attached to transitions as illustrations. An example super-state is marked. Within the superstate, all states are in equilibrium, their probabilities summing to 1. Transitions from the superstate to itself will change the equilibrium concentrations, but not the constituents of the solution.

Our model abstracts the details of the underlying concentrations of materials in the tube, instead it considers different states by the species constituting the solution. We may symbolically remove some constituents when considering reactions that have a negligible probability of reversing at the constant temperature the experiment is considered at, e.g fully hybridized instruction-input pairs such

as $a\bar{a}$. However, nothing ever leaves the solution. Furthermore, our input actions may introduce new strands into the solution. Hence, if we were to consider each state as distinct whenever it contained a different number of molecules, even when the species remained the same, we would never arrive at the same state again - we would have a cycle-free automata. In this case, the underlying probability space would be a Markov chain, albeit not a very useful one because it would never return to the same state. Also, there would be an explosion of states.

Instead, we extend the nondeterministic automata shown above with information about the distribution of constituents in each state. This extension takes a familiar form for those versed in probabilistic automata: It is the *adversary* (or *policy* or *scheduler*). The adversary is a function that resolves the non-determinism of an automaton by assigning a measure, consistent with the axioms of probability, to each *path* - an ordered sequence of states visited and inputs into the machine. We start by specifying the states. These can be uniquely and explicitly denoted by their constituent species and labeled for conciseness. For instance, the state at the top-right of Figure 3.3, consisting of the stack with instructions a and b, an unbound clock strand and an unbound input strand $\bar{a}$ is written

$$\sigma_6 = \{|ab, t, \bar{a}\}$$

A transition or input between states is just the set of input strands provided, e.g $\{t\}$ or $\{\bar{a}, \bar{b}\}$. A *path* is an ordered sequence of states and inputs, beginning and ending in a state. For example a path leading to the state σ_6 defined above, in Figure 3.3, is

$$\tau = (\{|ab\}, \{t\}, \{|ab, t\}, \{\bar{a}\}, \{|ab, t, \bar{a}\})$$

From a probability theory point of view, we restrict ourselves to finite input (and stack component) alphabets and finite-length stacks. Thus the space of possible inputs and space of states are both discrete and finite. This limitation is not grave as many of the practical computations we'd like to perform are finite in nature. An extension to a countably infinite state space requires a standard but somewhat more sophisticated application of measure theoretic arguments like the use of extension theorems. It is discussed in the specific context of probabilistic automata in a review paper by Stoelinga [**21**], and we do not address it here. The probability space of the adversary, Π, is the space of possibly infinite sequences of states and inputs. Standard constructions show that this is consistent with the requirements of a probability space.

For instance, an adversary Π that is consistent with Figure 3.3 above would have

$$\Pi(\{|ab\}, \{t\})(x) = \begin{cases} 0.002 & x = \{|ab, t\} \\ 0.749 & x = \{|b, a\} \\ 0.249 & x = \{a, b\} \end{cases}$$

Note that in general the adversary depends on the entire path, i.e the history of all inputs and states visited, and hence is not necessarily Markov. In fact it is trivial to see that in our case it will not be Markov: if we return from the state $\{|ab, t\}$ to itself by inputting more and more t strands, the reaction will be driven strongly towards the clock strand reacting with the stack to release the instructions, ending up at states $\{|b, a\}$ and $\{a, b\}$, so that the probability of state

$\{|ab, t\}$ becomes lower and lower each time. In other words, looking at the state space as a stochastic process, with $P_\Pi(\sigma|\tau)$ denoting the probability of ending with state σ after taking the truncated path τ (a truncated path is a path that ends with an action, not a state,)

$$P_\Pi(\{|ab, t\} \mid (\{|ab\}, \{t\})) \neq P_\Pi(\{|ab, t\} \mid (\{|ab\}, \{t\}, \{|ab, t\}, \{t\}))$$

That is, the path-independence (Markov) property often doesn't hold.

An additional notation introduced for clarity is marking all the states which result from an input to some given state as belonging to the same *super-state*. This is similar to the grouping done when transforming non-deterministic automata to their equivalent deterministic counterparts. The probabilities of all states in a super-state sum up to one, and it is sometimes useful to observe the behavior of the system in the super-state graph rather than the state graph to extract information about all potential species found at some step of the computation. For example, in Figure 3.2, the action t moving from the super-state to itself would correspond to having t actions outgoing and incoming from every state in the super-state. This is summarized with a single action at the super-state level.

We have arrived at a description of our clocked DNA program as a probabilistic automaton, and can now benefit from the extensive literature and results on the subject. An open question we do not treat here is whether the converse is true, that is whether every finite (probabilistic) automaton can be modeled using a clocked DNA program. In the results section we describe a DNA program for evaluating finite propositional logic formulas, which is certainly one interesting problem solved by finite automata.

However, we have not yet discussed what kind of adversary to use, or in other words what will realistically be the distribution of states we expect to see in the experiment. For that we need to introduce another notion into our model.

3.2.2. Molecular Concentrations. Thus far, the model we described lacks a key determinant of the actual equilibrium distribution of the reactants and productions in a chemical reaction: their molar concentrations. If we know the concentrations of the constituents of each state and the concentration with which we add each input, we can derive an adversary. We do that using the rate equations for all possible reactions, which gives us the distribution of next states - we have a way to resolve the non-determinism of the system. However, we will need to first extend our definition of the adversary to include concentrations. To do so, we order our finite alphabet of inputs Σ arbitrarily and denote this ordered set $\mathcal{S}$, with $N = |\mathcal{S}|$. We now consider paths whose states are the same but whose actions are replaced with vectors in $\mathbb{R}_+^N$, the N-dimensional real space with positive coordinates. Such an *action vector* x with C_i in its ith coordinate means we input the i'th strand of $\mathcal{S}$ such that its concentration in the solution increases by C_i times the initial concentration of the components of the first state in the path. The key here is that concentration **ratios** of products and reactants are all that matters to determine the equilibrium concentrations in the following state, not the absolute values. So we can arbitrarily fix the initial concentration of all components of the initial states to 1. The initial concentration in an actual computation using the model can then be specified at any value and with any units without hindering the general applicability of the results.

This does not limit us because we may have the initial state be a state containing only some virtual inert molecule with a reference concentration, arbitrarily chosen to be 1 in some units. That virtual molecule has the property that it cannot interact with any other species ever produced by our state machine. Now we can add the stacks as possible input words of our alphabet, and pour them at the quantities desired as inputs of the first step, using the general action vector notation. Since the stacks are designed so as not to react with one another we only increase the size of our state space by one as a result of this, and the probability to arrive at the desired state will always be one. This allows us to have arbitrary, unique concentrations for all species in our initial state by introducing a single extra state to the program.

A more significant limitation that our extension imposes is that we now have to consider a different and much larger probability space, that is harder to work with. We must now consider a space generated by the cones[1] over finite *extended paths*, those composed of states and action vectors in $\mathbb{R}_+^N$, beginning and ending at a state. Note that the underlying graphical model is still the same, because an input described by the vector x corresponds to the input set $\{s_i \in \mathcal{S} : x_i > 0\}$, and all vectors x with the same input set will lead to the same states. But the resulting distribution between all states in the super-state will be different, depending on the values of the coordinates of the different action vectors.

Note that we can now arrive at our previous description of the model by using only action vectors whose coordinates are either 1 or 0. Hence our extended model can be treated as a generalization of the simpler model that ignores concentrations. In fact we will describe example problems in chapter 5 using the simpler model - we keep the concentrations in mind only to accurately simulate an experiment and anticipate its results.

3.2.3. Gated stacks. Using the language we have covered so far we cannot implement very interesting computations. The stacks can unwind to sequentially introduce free instructions to the solution, and these may bind or not bind to various inputs. But essentially after enough clock strands have been introduced we arrive at a state where all stacks have unwound and the computation boils down to the result of throwing in all instructions and all inputs together at once into the solution. Interestingly enough there are questions this can answer, such as simple variations of the recognition problem described in Chapter 5. It is, however, in some ways analogous to writing a procedural computer program with no branches or loops (or recursion, for that matter) – which for the most part rules out many interesting applications.

We recall the gating mechanism described in Chapter 1. These gates allow for the equivalent of "if" statements in most computer programs as well as a rudimentary version of functions receiving no parameters and looping a predetermined number of times [**3**]. We use these gates to demonstrate some interesting computations afforded by our extended probabilistic automaton model together with the gated stacks.

[1]By cones we refer to the standard construction in countably infinite product spaces where an element of the basis for the set of measurable sets in the space is the set of all elements whose first i coordinates are a fixed prefix and the rest of the coordinates are taken to be the entire space of the coordinate.

A gated stack will be denoted by a 'dagger' (†) followed by the gate domain. Thus, $\dagger\bar{a}|bc$ means a stack containing the instructions b and c that is gated by the domain $\bar{a}$. Adding clock instructions will not do anything to the stack until an a domain has been introduced, at which point the gate on the stack will be removed from the state notation (as it can no longer react in ordinary circumstances) and we will remain with $|bc$. The choice we made to remove any domains that have undergone reaction from our state notation now pays off as we can have cycles in our state graph from states with gated stacks to states containing non-gated stacks of the same type (along with some remaining gated stacks, of course).

Stacks may have several gates and it is necessary to remove each of the gate instructions in order: $\dagger\bar{a}\bar{b}|ab$ is a stack that has the interesting property that with enough clock instructions, it will continuously unwind copies of itself until they are all exhausted. It releases a instructions to bind with the $\dagger\bar{a}$ gate, b instructions to bind with the $\dagger\bar{b}$ gate, removing itself completely from further reaction in the process. At the end, we remain with a and b instructions only, in the same quantity as was initially inserted to trigger the reaction. This is a form of non-explosive chain reaction.

3.2.4. Extracting output. A natural question about practically executing experiments with these models of computation is how to extract the output of the computation from the mixture where it had proceeded. While this is slightly out of the scope of our discussion, we do offer some options to the interested reader. First of all, at least some of the interest in biological computation and specifically DNA computation models has as its goals targeted pharmaceuticals and target-specific remediation. These do not require any output to be extracted and require only that the computation end with high reliability at the required state based on the inputs of the environment it is running in. That environment might be, for instance, a cell *in vivo*, as in Kossoy et al. [**10**] where a colony of bacteria develops different phenotypes based on a computation occurring inside the cells.

To monitor the results of the experiment *in vitro* we have considered several methods and found FRET to be the most likely to be useful in reliably providing information about the result of a computation. FRET, which stands for Fluorescent Resonant Energy Transfer involves two molecules, one containing a fluorescent donor and the other an acceptor, also called a receiver. When the molecules are very close to one another (on the order of 10nm), and excited with an external light source at the correct wavelength, they will emit detectable light, at a different wavelength than that which is emitted when the molecules are far away. Due to the relatively low concentrations used in our DNA processes, distinct molecules will get close enough for the FRET phenomenon to occur only when they hybridize. They are detectable even at concentrations of just 200,000 molecules per mL[**4**]. A typical procedure for extracting output using FRET will be to mark one or several of the instruction domains used in the computation with FRET acceptors, run the computation, and after it completes introduce the complementary domains modified to contain FRET donors (these are sometimes referred to as *flouromers*). When we then excite the fluorescent donors with light, we will see fluorescence only at the receptors' wavelength if there is a high concentration of them that are not bound to other molecules: So we measure the concentration of instructions that were released but did not hybridize with anything and remain single-stranded. We can release

fluorescent probes with different domains sequentially, measuring fluorescent output after each release. This way, we learn of each possible output in order. We may also fit the instructions with different fluorescent markers so that we have multiple-wavelength outputs that can be measured simultaneously. This is of course limited by the availability of flouromers with different wavelengths. Knowledge of the underlying probabilistic automaton is useful in that it informs us of which possible outputs need be measured to differentiate between different ending states.

Another viable option is gel electrophoresis. In this technique we extract all DNA molecules from the solution at the end of the experiment. Using radioactive film and a denaturing gel through which a current is passed, we receive indication of the approximate lengths (in nucleotides) and quantities of molecules occurring in the solution. To differentiate between several results in this case, we need to design the computation such that the output has varying lengths based on the result of the computation.

A final possibility is using iron balls coated with strands complementary to one of the expected output strands. We weigh the balls separately, then introduce them into the solution and, using a magnet, secure the balls while we wash out the rest of the solution. If a strand complementing the DNA sequences the balls are coated with is found in the solution, it will hybridize with its complement on the surface of the balls, and not be washed out. The iron balls will then weigh more than they did before the experiment. One may even use several sets of balls, specific to different sequences, if careful controls are employed.

In the example problems we demonstrate solutions using our DNA computer model, we utilize FRET as an example of extracting output. The reasons for this are that it is simple to use and very accurate, but also that it allows one to make several assays without disrupting the experiment. Much like the toy example given in Chapter 1, we can observe the output at one stage of the experiment and then continue, whereas with iron balls or gel electrophoresis we generally destroy the solution as a result.

CHAPTER 4

Simulation algorithms

4.1. Implementation of a compiler

In this chapter we describe the actual tangible product of our theoretical work, which is a "compiler" for computations expressed in the model of clocked DNA programs described thus far. In computer science, a compiler has several functions, the chief of which being turning algorithms expressed in an abstract and high level computer language into machine-specific instructions for executing the algorithm on a specified architecture and operating system; But a secondary function of the compiler is also to assist the programmer in ensuring his program is correct, for example using type systems. Our software, then, is somewhat analogous to a compiler: We take a high-level description of the program to be executed, design the specifics of the experiment that should be run such as the specific DNA sequences to use, and then provide insight into the expected behavior of the computation. By the last part we mean that, since we cannot actually execute the experiment on a digital computer, we instead simulate the computational process and provide statistical summaries of the margins of error, the states that the computation undergoes and the expected output. This verifies that the code indeed behaves in solution as its creator intended, at least in accordance with the laws of the simulator if not those of chemistry (though they are intended to match each other for the cases we deal with).

4.1.1. DNA Programs. Input into the compiler is provided in the form of a file containing a description of the constituents to be added into the solution at each input step and their concentrations. Thus it may specify we start with the state $\{|ab, |cd\}$ at concentrations $(1.0, 1.0)$ and then compute by pouring $\{t\}$ in concentration (2.0) where t denotes tick and tock strands. Internally, we represent the program as a list of super-states. Each super-state contains a set of states, and each of the states contains a set of all molecules in the solution. Concentrations are tracked for each of the constituent molecules in a hash table, and are updated as we progress in reading the input steps.

The output of the compiler consists of several parts. We output sequences for all domains, including the implicit clock domains. The sequences are selected to ensure the correct binding between them occurs so that the desired complexes form at the desired temperature and salinity conditions. We also output the heat map plot shown in Figure 2.1, depicting the probabilities of the sequences binding to each other in the intended way, to verify the specifics of the design. These outputs are generated by transforming the program from its representation as a series of domain instructions above to its representation as a series of bases with an unpseudoknotted hybridization pattern, feeding this representation as input for the NUPACK algorithms, and reading back the output and plotting it as necessary.

Because of the relative simplicity of this task and its reliance on previous work we do not dwell on its implementation nor prove its performance, which is dominated by the algorithms NUPACK employs.

The other output we provide is a graph, similar to Figures 3.1 and 3.2, of the pathway that the program takes with the output distributions of states. This graph is a path in the probabilistic finite automata implicitly defined by all of the stacks, instructions and possible inputs. The specific path described, then, comprises of just the actions defined in the program file. To compare different executions paths, one must provide a file with alternate input. This is preferable to an interactive approach because of its modularity and potential to be easily integrated into a larger automated software suite.

4.1.2. Programming model. The program is provided as single ASCII file. In that file, the ASCII notation for our model, described in Chang and Shasha [**3**] is used. In that notation, domains are written as their labels, while complements are denoted with stars (so $\bar{a}$ becomes `a*`.) Stacks are denoted with a 'pipe' character and gates with the 'plus' character: the gated stack $\dagger\bar{a}b|c\bar{d}$ becomes `+a*b|cd`. A reserved symbol, `t`, is used for denoting clock strands. We do not currently support strands composed of several domains, such as $a - b$, so no notation is provided for these.

The file describes the inputs into the solution. The entire program is a sequence of inputs, the first set of inputs corresponding to the initial state. Each input may be followed by a floating point number which will be used as the concentration to introduce it with. If the concentration is not specified, a default of 1.0 is used. The underlying format is YAML, described at `http://www.yaml.org/spec/1.2/spec.html`. The file is a sequence of inputs, and each input is a sequence of constituents. For example, the program beginning at $\{\dagger a|bc, \dagger b|a\bar{c}\}$ and followed by the inputs t, $\{\bar{a}\bar{b}\}$ where the clock is introduced in a concentration of four times as much as the other constituents will be written as:

```
-
    - '+a|bc'
    - '+b|ac*'
-
    t:  4.0
-
    - a*
    - b*
```

4.2. Algorithms for describing the state space of a program

The graph of the state space is created in a straightforward manner: We maintain a set of current states, starting with the initial state. Then, for every input step in the program file, we apply the ReactWithInput(S, I) function with the S being the last set of states it returned to us and I being the current set of inputs. This tail recursion discovers all possible outcomes of all possible reactions and populates the state space with it.

Procedure 1 ReactWithInput(S, I)

```
1: for s ∈ S do
2:     s ← s ∪ I
3: end for
4: repeat
5:     for s ∈ S do
6:         S_new ← S
7:         S ← S ∪ StateReactions(s)
8:     end for
9: until |S| = |S_new|
10: return S_new
```

Lines 4-9 constitute the "convergence loop". We generate more and more states by letting the constituents of the state to react with each other. This is important when for example we unwind a stack, and the instruction that gets released further reacts with another input of the state. We terminate the loop once there was no change in the states generated between this and the previous iteration, because it means we are in equilibrium: If nothing changed in the contents of the states from the previous iteration to the current, there is nothing new that can cause further reactions in the next iteration. The nature of the actual underlying chemical equilibrium may be dynamic, but inasmuch as the species involved are concerned, this is a correct representation of the equilibrium. It means we have fully explored the state graph reaching out from the initial states S with the inputs I, and discovered that all reactions now loop back into previously discovered states or there are no further reactions possible.

We must show that this iterative process is guaranteed to terminate. One argument is that we start with a finite number of states S and a finite number of input molecules I, each state containing a finite number of constituents which, if they are stacks, have a finite number of instructions. There is no room for infinity to emerge: Stacks become shorter with further reactions, and the solution gains exactly one instruction for each one a stack loses. Other constituents are instructions or input strands - these either do not react and so do not change or do react and are removed from solution, as they become unavailable for further reaction. Thus the space of states is bounded, specifically by $|S| \cdot M^K \cdot 2^{(|I|+KM)^2}$ where M is the maximal stack length (number of instructions plus number of gates) and K is the maximal number of stacks per state (which corresponds to the number of stacks in the input plus number of stacks in the initial state, since additional stacks are never created as a result of a reaction). The first term, M^K comes from all the states of unwinding for K stacks of length M. The second, $2^{(|I|+KM)^2}$ is an overestimate of all possible interactions between single domains released from the stack or found in inputs: Each either reacts and disappears, or doesn't react. So we have a choice of reacting or not reacting in the space of pairs. Finally we multiply everything by the number of states we can start from, the initial $|S|$. This expression also bounds the number of iterations. We are guaranteed not to be stuck in a non-converging state because we solely employ set unions, which means $|S|$ (in the pseudocode) either grows or stays the same. This also means $|S_{new}| \leq |S|$ is an invariant throughout the procedure. Specifically it is an invariant in line 9, and

as $|S|$ is a bounded natural number, and in line 6 $|S_{new}| = |S|$, we have that they must become equal at some point.

Procedure 2 StateReactions(s)

```
1: S ← {s}
2: for {c1, c2} ∈ s do
3:   T ← S
4:   for r ∈ Reation(c1, c2) do
5:     for t ∈ S do
6:       T ← T ∪ {s \ {c1, c2} ∪ {r}}
7:     end for
8:   end for
9:   S ← T
10: end for
11: return S
```

In StateReactions we create a state set that is the result of all possible interactions between two different constituents of the state. The iteration in line 2 takes all (unordered) pairs from the state. The Reaction procedure returns a set of all possible results of a reaction between c_1 and c_2. This means it always returns c_1 and c_2 as an unreacted pair, as well as all possible results of the reaction. We use this to create copies of the original state, without the reactants but with the products of the reaction in line 6. The end result is similar to a cartesian product of all possible reactions of two constituents, but where each resulting vector is treated as a set - a single possible resulting state. Mathematically, then, we produce the following:

$$\left\{ \{t : t \in v\} : v \in \prod_{\{c_1,c_2\}\in s} \textsc{Reaction}(c_1, c_2) \right\}$$

Procedure 3 Reaction(c_1, c_2)

```
1: if (c1, c2) are (Ungated Stack, Tick) then
2:   return {t_i : t_i = {l_1, ..., l_{i-1}, s_i},
                s_i is the stack without its i top instructions,
                l_i is the i'th instruction on the stack,
                0 ≤ i ≤ |Stack|}
3: else if (c1, c2) are (Instruction, Gated Stack) and the instruction complements
   the top gate on the stack then
4:   return {{c1, c2}, {Ungated stack}}
5: else if (c1, c2) are (Instruction, Instruction) and they complement each other
   then
6:   return {{c1, c2}, ∅}
7: else
8:   return {{c1, c2}}
9: end if
```

Finally, the Reaction method is a description of our semantics of interactions between molecules: A gated stack can interact with an instruction strand to remove

that gate, if they are complementary. Two instructions (or input strands, for that matter) that are complementary, may be considered to be removed. A clock strand interacting with a stack will return all states of unwinding for the stack, from remaining unwound to completely dissolving into its constituent instructions, but the clock strand itself is always removed from the solution. This introduces the states corresponding to "runaway" stack unwinding. Finally, if none of the cases hold, there is no reaction - returning the same pair of molecules will cause us to leave the state unchanged. We use pair notation here (as in "(c_1, c_2) are (Ungated stack, Tick)") to mean that one of c_1, c_2 is an ungated stack and the other is a tick strand. The reactions are symmetric and we do not care which is which – we use this notation only to simplify the pseudocode.

4.2.1. Complexity. The complexity of the REACTION procedure is determined by the size of the set it returns. The creation of the set that results from the reaction of an ungated stack and clock strands dominates all other possible return values. This set contains $O(|\text{Stack}|)$ elements, so its complexity across all possible stacks in the program could be bounded by $O(M)$ where M is the maximal length, in instructions, of any stack in the program (either input or one of the initial stacks).

The STATEREACTIONS procedure takes every pair of constituents of the initial state, and for every possible reaction between them, updates all previously created states. As noted in the description, this is similar to calculating the cartesian product. The number of operations is on the order of $O(M \cdot K^4)$, where $K = |s|$, the size of the input state. A K^2 factor comes in from iterating over all pairs of constituents, and for each iteration we add all of the possible $O(M)$ outputs from REACTION to T. T grows from 1 to the full cartesian product which is certainly bounded by $O(M \cdot K^2)$, so we iterate at most this many times for each pair of constituents. The state set T and each of the states in it are sets represented as hash tables. However, since the program can be read entirely and the input alphabet precomputed as a preprocessing step, we can create a perfect hash, so that we can add and remove entries from s and T in $O(1)$ time.

Looking at REACTWITHINPUT, we note that despite the fact that the theoretical bound on the number of convergence iterations is astronomically large, in practice almost all programs converge within 1-3 iterations. The reason is that for non-convergence to be maintained, you have to constantly introduce new states into the solution. But new states are created only by unwinding stacks (releasing instructions) or unlocking gates. The typical reaction requiring the convergence loop is one in which a clock strand releases instructions, which later react among themselves or with gated stacks. All instructions that can possibly unwind and all gates that can possibly be removed do so in one call to STATEREACTIONS. We see that this way the exponentially-large state tree is being explored layer by layer, and the number of layers is of course logarithmic in the size of the tree (the branching factor here is almost always larger than 1, as is evident from the return values of REACTION.) The longest chain of reactions that can actually occur is one in which several instructions are released due to clock strands or directly input in the solution, followed by sequential ungating of all stacks by these instructions, exposing new gates. This proceeds at a rate of one gate being removed from all of the stacks at every iteration, which means you will not see in practice more convergence iterations than the maximal number of gates on any stack, which is certainly bounded by the size

of the largest stack, M. So we have $O(M)$ calls to StateReactions (this includes the internal loop), and that is the largest of the terms for ReactWithInput's time complexity. Combining this with StateReactions' complexity we get to a grand total of $O(M^2 \cdot K^4 \cdot |S|)$, where I've rolled into K the terms $K + |I|$ representing the size of a state. In practice M^2K^4 serves as a constant "program size" and is relatively small. $|S|$ begins at 1, the sole state being the initial state in the program, but grows as we add inputs in stages to the solution. Subsequent calls to ReactWithInput are made with the S_{new} received from previous calls, until input is consumed. The tracing of the state graph for an entire program with D input steps, then, can be thought of as an exponential $O(M^2K^4 \cdot \phi^D)$ where ϕ is the average branching factor, bounded on the number of states. In the first call we have 1 state and we can certainly have no more than $M^K \cdot 2^{(|I|+KM)^2}$ states at the end.

4.2.2. Simulation. There are various approaches to biochemical simulations. By and large, these can be divided into 3 categories: stochastic, approximate stochastic, and hybrid[**13**]. All of these methods are relevant whenever exact methods, relying on algebraic solutions to the equations that govern the system are either insoluble or have no known solution. Also, when the number of molecules is relatively small, so that these equations which describe the expectation rather than the law might not hold because of small population effects, we will wish to use stochastic simulations. In the latter case the behavior can vary widely due to random fluctuations, and samples from the range of the possible outcomes, which provide confidence intervals instead of exact values, are of interest (e.g to determine thresholds). Many software packages provide these kinds of simulations, and VisualDSD specifically does so for simulating DNA interactions. However, its closed source nature has somewhat restricted us from interacting with it in a manner favorable to us, and since we are modeling behavior based on millions of complexes at once (i.e the thermodynamic limit behavior rather than small-population dynamics), we instead have used NUPACK. NUPACK gives us the partition function and free energies for each resulting state, allowing us to report the fraction of the molecules in the mixture that will result in each state, specifically in the states corresponding to the desired output. Our intention is that the researcher may then look at the output and decide if the fraction of correct computations is significant enough for his purposes, and if it is not, tweak his mechanism (the underlying state machine, gated stack composition and so forth), or the various parameters of the computation such as temperature, lengths of instructions and clock domains, and the salinity of the solution in order to derive more favorable conditions. The following chapter deals with an example class of computations that might be of interest to researchers, defined using our model. When these are input as programs to our compiler, it will instruct the researcher on what are the exact sequences to be used for each of the domains (possibly fixing certain sequences to ones specified by the researcher), and what will the distribution of final states look like.

CHAPTER 5

Results and example clocked DNA programs

In this chapter we will give example problems that are solved by our model easily and are otherwise difficult to construct an experiment for. These problems were chosen for benefiting from the availability of synchronous operations, branching and procedural programming in our model.

5.1. Strand recognition

The strand recognition problem is one in which we would like to differentiate between possible input strands, detecting the presence or absence of certain domains. The very simplest case is detecting whether a certain domain, a, is present or not in input strands. For example, one might imagine a to be a short sequence of DNA encoding for a particular harmful property - perhaps a mutation in a certain protein, or a region with some other meaning we would like to detect. We have several tubes containing DNA strands from different sources, and we would like to be able to say which, if any, of these tubes contains strands with the specific domain we are looking for.

An experimenter that would like to determine this without using our clocked DNA model faces no big problem: She may, for example, add to each tube the complementary $\bar{a}$ domain in the form of a strand that she synthesized. That population of strands should all be taken so that they are of the same length (in bases). In a control tube she will keep just that strand $\bar{a}$, by itself. Wherever the a domain existed in the strands already in one of the solutions, the introduced strand containing the $\bar{a}$ domain will hybridize and form the stable double-stranded DNA complex $a\bar{a}$. She can now either employ iron balls binding $\bar{a}$ domains (i.e coated with a strands) and see that they did not bind any $\bar{a}$ in the solution, by weighing the balls, or she can apply gel electrophoresis to each of the mixtures and detect a missing band compared to the control solution.

We note that a solution involving fluorescence has a drawback in this scenario: Using a modified $\bar{a}$ strand containing a fluorescent acceptor molecule may inhibit to some degree its hybridization with the unknown strands a that we wished to detect in the tube. Results derived in this manner, then, could be slightly inaccurate if used to measure rates of hybridization, for example. The clocked DNA program solution avoids this problem and allows us to use a fluorescence signal which is more accurate, and easier to apply experimentally compared to methods like iron balls and electrophoresis, and allows for a continuous signal over time - a capability we later show can be exploited by the clocked computation to derive output unattainable in other systems.

When the target strand to be detected does not consist of a single domain, but rather a boolean logic propositional expression P over possible domains, such as "find an a or a b, but not an a and a c", the experimenter using naive techniques

such as the one above has a much bigger problem. In fact, to answer exactly the question above one would classically be forced to split his original mixture into at least two separate tubes. He will then test for the presence of a in one, and b or c in the other, and combine the answers from both in order to decide the value of the original propositional logic formula P. There are variations of the experiment above utilizing different lengths of strands, balls coated differently or multiple fluorescent markers that would be able to achieve this in a single tube. But as the number of domains involved in the expression grows so does either the complexity of the experimental procedure, the resolution of instruments required, or the number of tubes and control assays. In comparison, a clocked program utilizing branches and synchronization is an ideal system to easily model these problems. In pseudocode we may write the predicate above as

Predicate 4 $(a \vee b) \wedge \neg(a \wedge c)$

```
1: p ← False
2: q ← True
3: if a then
4:     p ← True
5: end if
6: if b then
7:     p ← True
8: end if
9: if a then
10:    if c then
11:        q ← False
12:    end if
13: end if
14: if p then
15:    if q then
16:        return True
17:    end if
18: end if
19: return False
```

This pseudocode is a convoluted way to express something most modern programming languages will let you express in one statement – but it is an accurate transcription of how our model will solve this problem. We will have each subexpression generate a "result" strand (equivalent to the variables p and q in the example above). We then combine the result strands in the same fashion we combine the underlying, base-level queries about domains in the solution such as a, b and c in this example. The final result strand (represented by **return** statements in the text above) will be a FRET acceptor DNA molecule. When we pour in its complementary DNA sequence with a fluorescent donor molecule into the solution, the existence or lack of fluorescence will inform us whether the entire expression evaluated to true or false.

We assume everything is in the solution prior to "executing" the program (by pouring in the clock strands). Another way of doing this is simply pouring the program into the solution containing the input DNA strands to be detected.

To demonstrate how we solve this specific expression and general propositional logic formulas in our model, we start by presenting a solution to detecting just a single domain, say a.

A very straightforward solution would be to have a stack containing a single instruction, $\bar{a}$ tagged with a fluorescent marker that will exhibit FRET when it binds to an output extraction molecule containing both the domain a and a fluorescence donor. This simple approach does not necessitate procedural programs or instruction stacks at all. It is very similar in nature to the trivial approach we initially described, and only artificially places the probe strand $\bar{a}$ inside of a stack instead of introducing it directly. As such, it also doesn't scale as the complexity of the propositional expression grows (indeed, detecting the presence of a and b together in the target strand but not either of them alone is already impossible with this approach). It also suffers from requiring that the domain-detecting molecule contain the fluorescent marker molecule, which may interfere with its regular binding to the target.

A more refined approach, then, would be to include both a stack containing the instruction $\bar{a}$ as well as another stack, gated by the domain a, and containing a flouromer with a separate domain, denoted f. This solution is portrayed in Figure 5.1 below. If the target strand in the solution contains the a domain, then once clock strands have released the $\bar{a}$ instruction to freely interact with other constituents, we expect that the instruction will bind to both the gate a and the target. The number of molecules bound to each of the two species is determined by the respective concentrations of the gated stack and the target. If we set both to be equal, thermodynamics dictates that half of the released $\bar{a}$ instructions will bind to the target and half will bind to the gate. Thus, once we introduce more clock strands into the solution to release the f instructions from the now ungated second stack, we expect to see half of the fluorescent signal that we would see in a solution without the a target strands (where all stacks containing f are ungated because there was no competitive binding).

We refer here to receiving half the fluorescent signal instead of specifying exact units because it is often the case when using spectroflourometry that units are arbitrarily set by measuring a ground state with no excitation (i.e in the dark) and a state with the maximal concentration of fluorescent molecules possible in the experiment. In dilute solutions the absorbed light is linear with the concentration of the fluorescent particles [**4**], so we can express fluorescence as a simple fraction of the maximal attainable in our experimental system.

Returning to our current approach, we find it is better in that it does not rely on a having a fluorescent receiver molecule, but it is still not scalable: The dependence on detecting the level of fluorescent signal will cause us to run into problems once we introduce more complex queries (propositional logic expressions consisting of more clauses and domains). For instance, we might want to trivially extend this by asking whether a or b is in the target strand. The amounts and concentrations of the various components would have to be carefully managed so we can differentiate the cases where both a and b were found in the target versus cases where either a or b were found or cases where neither were in the solution. In a naive implementation, the existence of both a and b will lead to seeing a quarter of the fluorescent signal. Either of them but not both will give half of the signal, while the maximal fluorescence achievable by the f domains released from

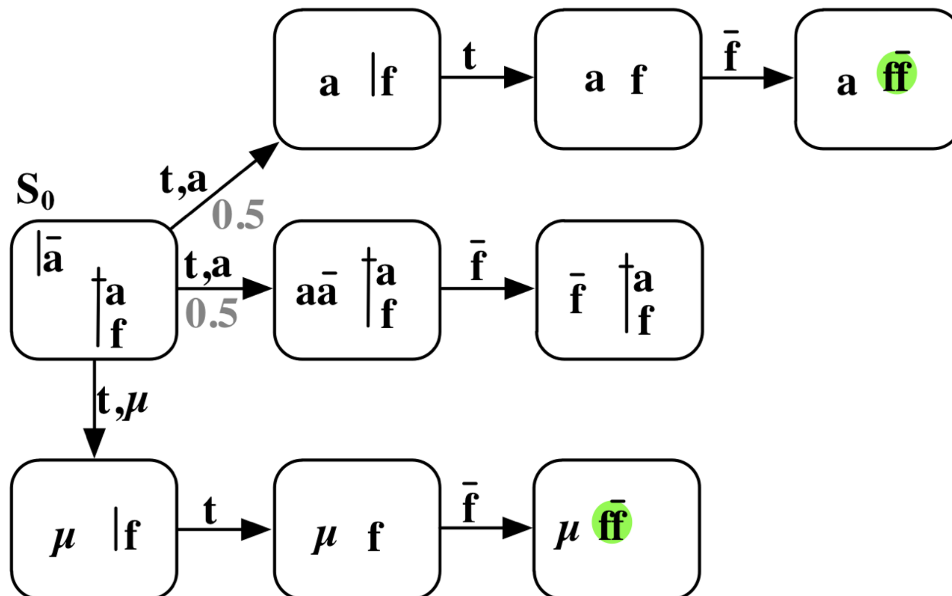


FIGURE 5.1. Detection of a domain without decoupling. Not all states depicted. The initial state is marked S_0. Gates on a stack are denoted by a tick mark on the stack bar immediately preceding them. Adding an a domain together with clock strands will cause half of the released $\bar{a}$ instructions to bind to the gate of the second stack, while half will bind to the a domain. As a result, adding the $\bar{f}$ flouromer will cause half the maximal level of fluorescence, compared to the other input pathway. The other input pathway corresponds to pouring any strand $\mu \neq a$. If that occurs, pouring clock strands will cause all of the $\bar{a}$ instructions released to bind to the gated stack, releasing the fluorescent receptor f. Upon subsequent addition of complementary fluoromers $\bar{f}$, we will get fluorescence. Thus full fluorescence signifies the logical expression a evaluated to false, while half fluorescence indicates a is true.

the stacks is emitted when neither is in the solution. This may sound appealing due to the amount of information that can be extracted from a single measurement of fluorescence – here, we have 3 possible results instead of a binary "true or false" output. However, it can in fact lead to practical problems in the resolution and accuracy of measurement instruments, as well as a complicated experiment design requiring careful control of the concentrations of various stacks, clock strand poured and so forth. Similar problems also arise whenever there are multiple occurrences of a in the target strand: For example under this naive implementation, two non-overlapping occurrences of the a domain in the target will cause only a third of the fluorescent signal to appear. We wish to remove the dependence on the level of fluorescent signal in interpreting the results. In a sense, we wish to digitize the signal as one does in electronics when moving from analog circuits to reliable digital ones.

5.1.1. Decoupling. All of these problems essentially stem from the lack of decoupling between the reactions allowing a to bind to the target if it exists, and the

reactions allowing a to bind to the gate if it doesn't exist. This causes a to "divide its attention", so to speak, between two competitive products having the same free energy and therefore the same rates of reaction. Thus we employ the power of the clocked DNA model to allow synchronizing the reactions so that they occur separately, using the following robust design for one-domain detection (Figure 5.2).

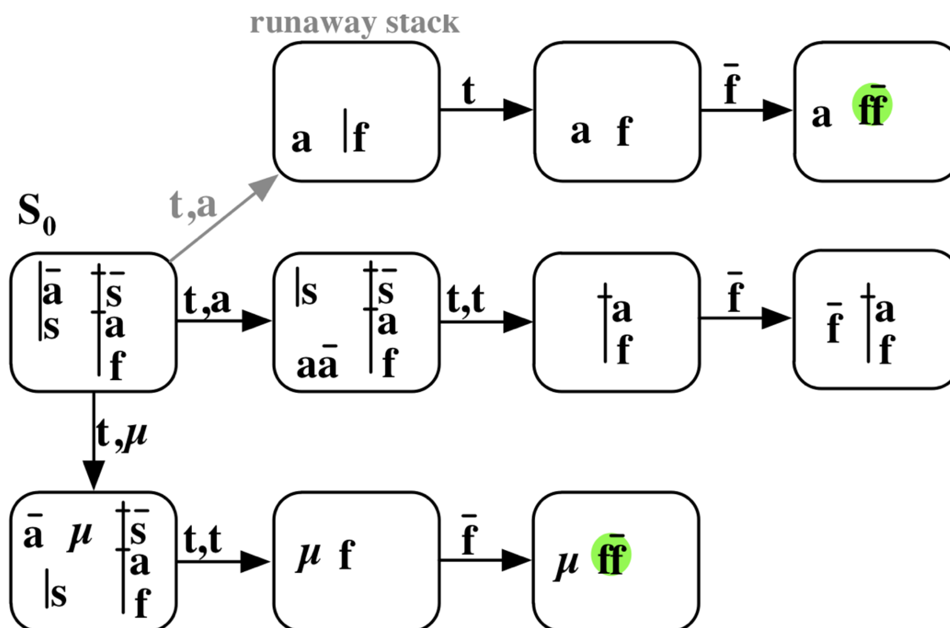


FIGURE 5.2. Detection of an a domain with "half" decoupling. Not all states are depicted. The initial state is marked S_0. Adding an a domain together with clock strands will cause most of the released $\bar{a}$ to bind to the input domain a. Rare runaway stacks will release s instructions, which will ungate the second stack ($\dagger\bar{s}a|f$), allowing rarer-still remaining $\bar{a}$ instructions to further ungate that stack, and lead to very slight fluorescence once $\bar{f}$ fluoromers are added. Adding $\mu \neq a$ domains instead of a domains will lead to the input pathway at the bottom of the figure, where clock strands cause gradual ungating of the stacks until all possible fluorescent donors (f instruction) are released, leading to full fluorescence when their complementary $\bar{f}$ fluoromers are added. Thus full fluorescence signifies the logical expression a evaluated to false, while very slight fluorescence can be treated as no fluorescence, signifying a is true.

To detect the presence of the a domain we have a stack containing two instructions: $\bar{a}$ (a's complement), and s, a special auxiliary *synchronization* instruction. s is guaranteed by our compiler to never interact with a or $\bar{a}$. We also have a gated stack, whose gates are the domain $\bar{s}$ complementary to s, followed by the domain a. That stack is our result stack, and it contains a fluorescent receiver attached to a sentinel instruction f.

What we have ensured by this is that the instruction f is released only after all $\bar{a}$ instructions have had the chance to bind to a domains in the input strand(s), if

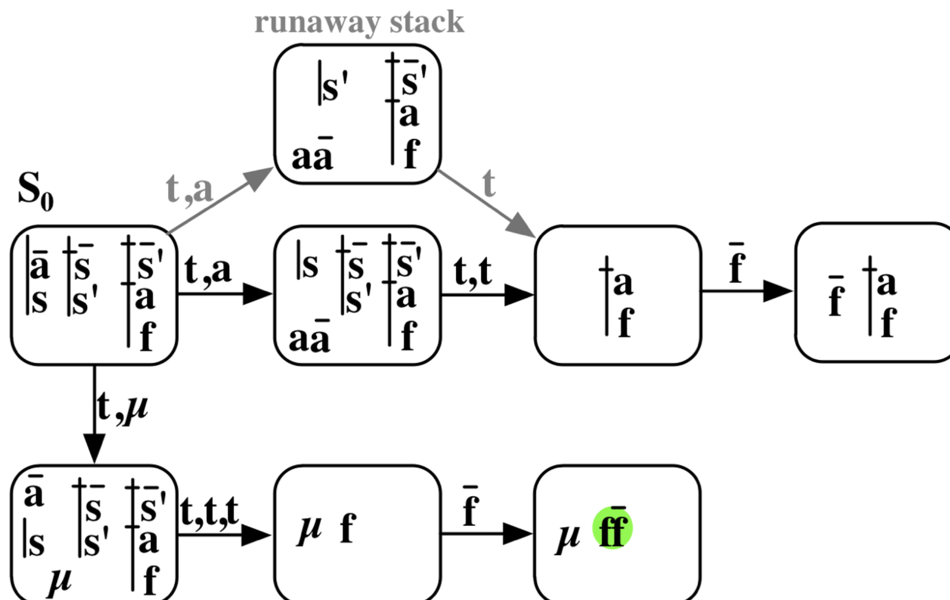


FIGURE 5.3. Fully decoupled detection of a domain: Runaway stacks are incapable of causing unwanted fluorescence. Again the initial state is marked as S_0, and here adding a with a clock strand will lead to either binding to the input a, or cause runaway stacks to ungate the second stack. However, more clock strands are necessary to ungate the third stack (the result stack), and therefore no fluorescent f strands are released. Instead, there is ample time for the released $\bar{a}$ instructions to react with a inputs until all of them hybridize correctly. Then, additional clock strands will ungate the remaining stacks, but no $\bar{a}$ instructions are left to ungate the final stack, which means no f strands are released, so no fluorescence occurs. Conversely if $\mu \neq a$ is introduced into the solution, clock strands will cause complete ungating of all stacks until all f strands are free, with the consequent addition of $\bar{f}$ fluoromers causing full fluorescence. Here there is no "leak" signal: Fluorescence, if it occurs, will always be entire and means the propositional expression (a in this case) was false. Conversely no fluorescence means the expression was true (a was present)

these exist in the solution. The reason this is ensured is that when the experimenter pours in the clock strands at a quantity less than or equal to the quantity of the first ungated stack, they release the $\bar{a}$ instruction, and to a small extent, the s instruction in the case of runaway stacks. The $\bar{a}$ instructions released are unable to bind to their complementary gate a, since it is blocked by the $\bar{s}$ gate preceding it. Instead, they bind to the input instruction containing a domains if it exists, or otherwise remain free in the solution. Some instructions s created by runaway stacks do bind to the $\bar{s}$ gate and remove it, allowing any remaining $\bar{a}$ instructions to bind to the following gate. But the rate of reactions governing the unwinding of a stack makes runaway stacks relatively rare. In addition, the extra reactions necessary to first bind the s gate, then disassociate the gate complex from the stack, thereby revealing the a gate, are both slow, with rates on the same order of

magnitude of binding the a domain. Thus there is an almost complete decoupling of the reactions.

In fact, complete decoupling (Figure 5.3) may be achieved by using one additional intermediate stack, gated by s and releasing s'. The released s' then controls the gate of the final result stack instead of the $\bar{s}$ gate (that is to say, we replace the $\bar{s}$ gate by $\bar{s'}$). Then, the addition of clock strands by the experimenter is necessary for the result stack to unwind, achieving a decoupling of the two reactions by an arbitrary time delay imposed by the experimenter. This is achieved because the rate of reaction for clock reactions versus instruction reactions (like binding to gates) is determined primarily by the length of the instruction domain, which is designed to be longer than the clock domain. The orders of magnitude difference in the rates of reaction for clocks and gates ensures that nearly all of the clock strands released will be consumed by releasing the $\bar{a}$ and s instructions, and furthermore that nearly all of the $\bar{a}$ instructions that could interact with free a domains in the solution will do so before binding to the a gate, because the s' reactions won't be released yet: There are no more available clock strands to release them, having all been used up quickly in unwinding the initial stack. Figures 5.1 - 5.3 above illustrate the difference between the levels of decoupling.

5.1.2. Half-decoupling output fluorescence levels. We now know what to expect of the "no decoupling" and "full decoupling" states in terms of the level of fluorescence expected as output in case a is in the solution, or it is not. For the half-decoupled setting, the equilibrium concentrations for the number of $\bar{a}$ instructions bound to the a gate and the number of $\bar{a}$ instructions bound to the target, if it exists, are governed by the following equations (assuming all concentrations of all constituents is the same):

$$[\bar{a}a] = [\bar{a}_0] \cdot \frac{k_1}{k_1 + k_2} \qquad\qquad [\text{Gate}_{\bar{a}a}] = [\bar{a}_0] \cdot \frac{k_2}{k_1 + k_2}$$

Where k_1 is the rate of reaction for the hybridization of $\bar{a}$ with its complement, discussed in Chapter 2, k_2 is the combined rate of reaction for the binding of s to the $\bar{s}$ gate and disassociation of that $\bar{s}s$ complex from the stack, and $[\bar{a}_0]$ is the concentration of $\bar{a}$ instructions existing in the solution. Note all of these $\bar{a}$ instructions are a result of runaway stacks, the probability of which is in turn determined by the rates of reaction of the mechanism of unwinding of the stacks. These rates can be determined experimentally, but by estimating the rates involved we believe that this decoupling is satisfactory for most cases. Furthermore, the complete decoupling outlined above guarantees that very few stacks will unwind unintentionally, the cost being an additional stack and additional strand species per domain in the propositional logic expression. Adding additional decoupling stacks (gated by s' and releasing s'' and so forth) adds virtually nothing to the isolation of reactions achieved by the full decoupling method.

One should note that, as we move towards more complicated cases involving several steps, each s domain is unique to the specific stack it appeared in – i.e it decouples only the binding of the instructions in that stack to the result stack. Two separate stacks used for detecting different strands will have distinct synchronization domains that do not interfere with each other.

We note the result instruction, f, is released if and only if the following two conditions occur: (1) $\bar{a}$ hybridized with the a gate on the result stack and (2) the

result stack had unwinded by subsequent pouring of the clock strand. Condition (2) occurs if and only if $\bar{a}$ had not first hybridized with an a domain on the input strand. So, in conclusion, fluorescence appears when there was *no* a domain in the input strands. This will remain a property of our computation and is important in order to reason about the behavior of the system: The absence of a result strand means the expression it acts as a result for evaluated to true. Its existence free in the solution indicates falseness of the expression. In a sense this is the reverse of the prevalent design principle of electronic circuits where the presence of voltage on a wire often indicates truth value. Otherwise the two systems are somewhat analogous. That analogy is elaborated in a later section.

We have discussed, then, detection and interpretation of the result for a single domain. We will now show how this serves as a building block for arbitrary finite propositional logic expressions.

5.1.3. OR stacks. The construction of arbitrary OR expressions, $d_1 \vee d_2 \vee \ldots \vee d_m$ uses a single result stack that has all of the domains d_i as gates one after the other. Thus, the resulting f fluorescent instruction is released if and only if all of the gates were opened, corresponding to a case where none of the d_i's matched the input domain. Again, full fluorescence always results from a false predicate. An example $a \vee b$ stack is demonstrated in Figure 5.4.

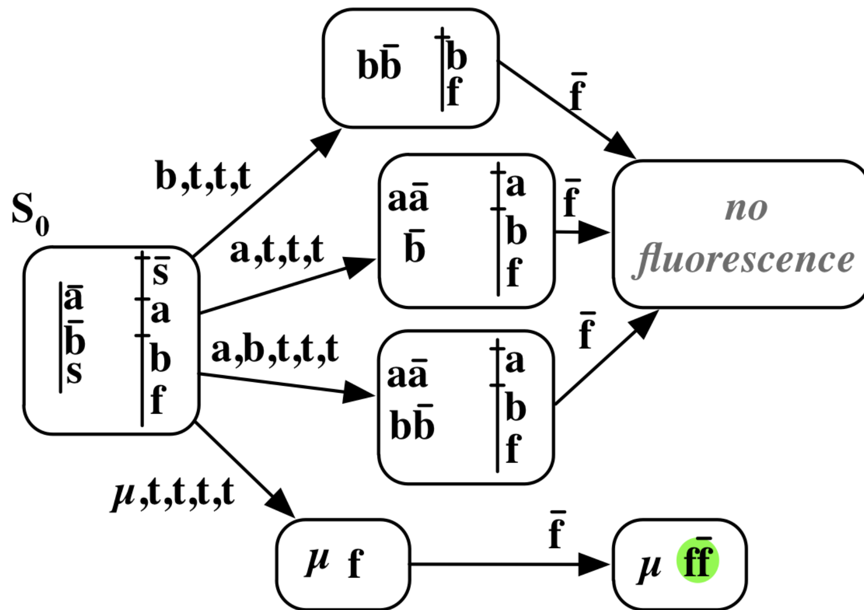


FIGURE 5.4. The configuration for solving $a \vee b$ with half-decoupling. This abbreviated graph does not show runaway stacks or incomplete transitions. Starting from S_0, the addition of a, b or both and enough clock strands will finish in a state where no f fluorescent receiver strands are free, signifying $a \vee b$ was true. Addition of $\mu \neq a, b$ instead causes the f stack to be released and fluorescence is emitted once $\bar{f}$ fluoromers are introduced into the solution. This means neither a nor b were in the solution.

5.1.4. AND stacks. The construction of arbitrary expressions of the form $d_1 \wedge d_2 \wedge \ldots \wedge d_m$ is by multiple result stacks: The first is gated by s and d_1, the second by s and d_2 and so forth. All stacks are present in the same concentration and thus full fluorescence occurs if any of them is ungated, signifying that one of the instructions $\bar{d}_i$ hybridized with the gate and not with input strands, which means the complementary domain is missing in the input strands, and the expression evaluates to false. Here the number of resulting fluorescent instructions is actually saturated by having more than one missing domain. However, since we only end up pouring a known quantity of the FRET donor molecule, fluorescence will still be at either the full level if any of the result stacks were ungated, or none at all if all domains were found on the target strand.

Note that both of these constructions are also able to detect overlapping regions in the input so long as the input strand is supplied in high enough concentration as to allow each possible d_i instruction to bind to a single input copy (hence, a concentration m times the concentration of the d_i stack should suffice).

In both the OR and the AND stacks, it is not important whether the d_i's lie in the same stack or on separate stacks. We shall assume they do lie in the same stack, as it simplifies the abstraction process when we combine them to form more complex stacks detecting composite queries.

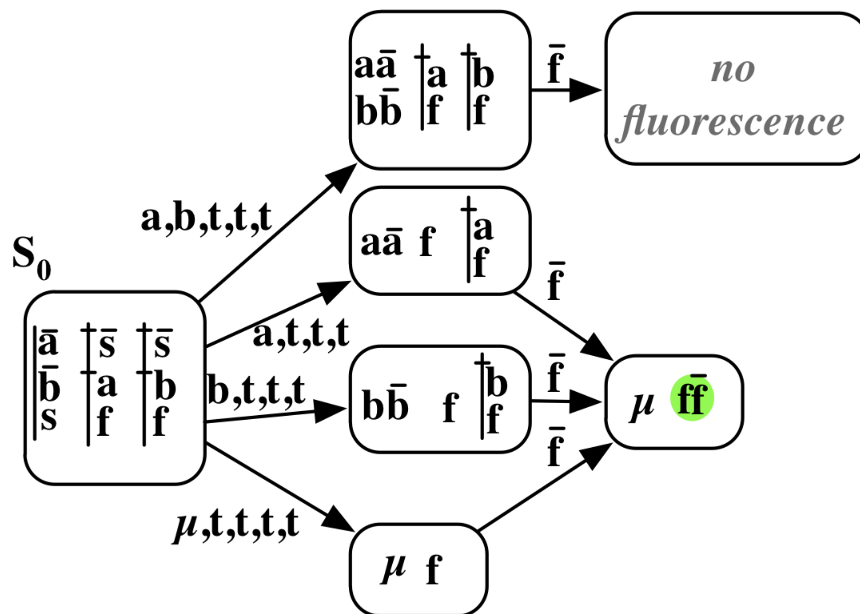


FIGURE 5.5. The configuration for solving $a \wedge b$ with half-decoupling. Again we show only most probable transitions. Starting from S_0, the addition of both a and b and enough clock strands will finish in a state where no f fluorescent receiver strands are free, which we interpret as $a \wedge b$ being true. Addition of anything else causes the f stack to be released and fluorescence is emitted once $\bar{f}$ fluoromers are introduced into the solution. This means we did not find both of the domains in the solution.

5.1.5. NOT stacks. A naive approach might have been to treat a logical not by simply reversing the way we interpret the output: I.e, treating the evidence of a FRET signal as meaning that the logical expression is true rather than false. However, this fails in all but the simplest of cases because once several domains are combined the effects of NOT have to be modeled in the component stacks themselves to affect the logical calculation, not only in our interpretation of the result. The stacks for $\neg d$, then, are implemented by adding the flourescent instruction f to the initial stack containing $\bar{d}$, and replacing the contents of the result stack by an *non*-fluorescent $\bar{f}$ instruction (i.e one that does not induce fluorescence, as it does not include a FRET donor molecule). Hence, whenever $\bar{d}$ binds to the gate of the result stack (when it did not find any input strands to bind to in the solution), this "quenching" $\bar{f}$ hybridizes with the f instruction, preventing further reaction with FRET donor molecules and thereby preventing fluorescence. The benefit is that we interpret the output as in the previous cases: The expression $\neg d$ is true, or in other words, d is *not* in the input strands if there is no fluorescence. If, however, d had been a domain in the input, $\bar{f}$ would not be released as nothing would ungate the result stack. The f instruction released from the initial stack would react with the $\bar{f}$ fluoromer we introduce when we wish to read the output, giving us the fluorescence which we interpret as meaning $\neg d$ is false, that is d is true.

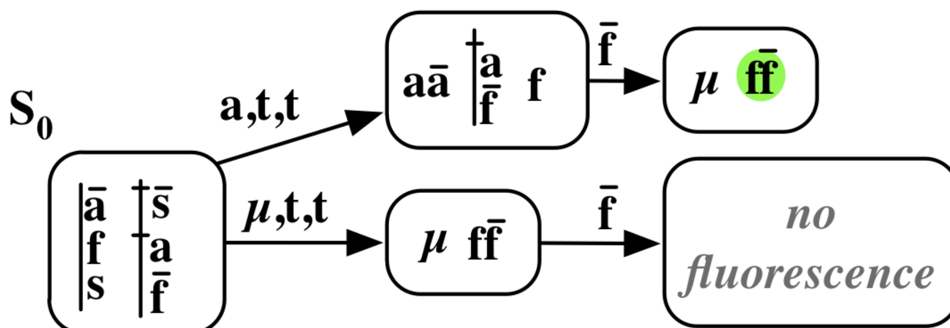


FIGURE 5.6. The constituents for the $\neg a$ query, with half-decoupling. We omitted the states that are not part of the main pathway demonstrating the solution. We start from S_0, where addition of clock strands will release an f instruction together with the "probe" domain $\bar{a}$. If a is not in the input, the $\bar{f}$ (non-fluorescent) domain will be released, hybridize with the f domain to form the stable $f\bar{f}$ complex, and no fluorescence will occur even after we introduce the fluoromer $\bar{f}$. If, however, a was in the input, the fluoromer $\bar{f}$ will bind to the f domain and the fluorescence emitted will let us know the proposition $\neg a$ had been false.

One trivially sees that this also works for negating stacks of the "or" or "and" types: Adding the f instruction in the initial stack and replacing the f instruction in the result stack with a non-fluorescent $\bar{f}$ instruction achieves the correct behavior.

5.1.6. Summary of the fundamental logical recognition components. We have described the components of a propositional logic expression in terms of the components of a clocked DNA program that can check if the expression is satisfied given a set of input variables coded as different DNA domains. In the process,

we loosely used the terms "initial stack" and "result stack". In moving forward to demonstrate how to combine these building blocks into arbitrary propositional logic expressions, we formalize these notions. The map F between a propositional logic expression P and the DNA program Ω used to satisfy it has the following properties:

$$P = d_1 \vee d_2 \vee \ldots \vee d_n \Rightarrow F(P) = \left\{ |\bar{d_1} \ldots \bar{d_n} s_P, \dagger \bar{s_P} d_1 \ldots d_n | f_P \right\}$$
$$P = d_1 \wedge d_2 \wedge \ldots \wedge d_n \Rightarrow F(P) = \left\{ |\bar{d_1} \ldots \bar{d_n} s_P, \dagger \bar{s} d_1 | f_P, \dagger \bar{s_P} d_2 | f_P, \ldots \dagger\ \bar{s_P} d_n | f_P \right\}$$
$$P = \neg d \Rightarrow F(P) = \left\{ |\bar{d} f_P s_P, \dagger \bar{s_P} d | \bar{f_P} \right\}$$
$$P = \neg(d_1 \vee d_2 \vee \ldots \vee d_n) \Rightarrow F(P) = \left\{ |\bar{d_1} \ldots \bar{d_n} f_P s_P, \dagger \bar{s_P} d_1 \ldots d_n | \bar{f_P} \right\}$$
$$P = \neg(d_1 \wedge d_2 \wedge \ldots \wedge d_n) \Rightarrow F(P) = \left\{ |\bar{d_1} \ldots \bar{d_n} f_P s_P, \dagger \bar{s_P} d_1 | \bar{f_P}, \dagger \bar{s_P} d_2 | \bar{f_P}, \ldots, \dagger \bar{s_P} d_n | \bar{f_P} \right\}$$

Where all domains involved (d_i, s_P, f_P for $1 \leq i \leq n$) are distinct and do not interact with each other (except with their own complementary domains). F gives us, for every simple proposition of the kinds described above, the constituents of the solution necessary to compute it. The result of F is a set of stacks in our notation, that will evaluate the expression using the mechanism we described. We wrote F here using half-decoupling, but one may trivially add the decoupling stacks necessary to establish full decoupling for F, keeping in mind that s and s' should be specific to the predicate P, as denoted above. The stacks containing the f_P strand are called the *result* stacks of the expression, the stack containing the domains d_i is the *initial* stack, and any intermediate stacks, if present, are *decoupling* stacks.

5.1.7. Composing subexpressions. The key to combining these expressions is that f_P does not have to be fluorescent in itself. It may instead become the input to further gates and hence determine the computation. In fact, this becomes a physical form of lazy evaluation – if f_S for a sub-expression S in P is not released, because that expression was satisfied, and in the case that expression is taken in a disjunction (i.e $P = S \vee T$), no further reactions in T are necessary, and indeed none will occur because the gates for the stacks with the logic for verifying T are not opened.

We now want to solve the problem of detecting whether a general logical propositional expression P over domains $d_1, \ldots d_m$ is satisfied by the inputs of a solution. In fact we will see we are able to do even more, and determine which of the sub-predicates contained in P matched the input strands. The importance of this result is in its practical application to experimenters: We have devised a system which enables an experimenter to decide among many possible features of DNA strands in the solution by simple repetitive pouring of clock strands and a single colored fluorescent marker, with no temperature cycling or washing steps and with readily interpretable output. To demonstrate this construction we will demonstrate it with the simple example predicate $(a \wedge b) \vee (c \wedge d)$.

We denote $S = a \wedge b$ and $T = c \wedge d$ consider only $F(S)$ and $F(T)$, but do not use fluorescent instructions f_S and f_T. Instead, we construct an "or-type" result stack for $S \vee T$, i.e an additional stack gated by f_T and f_S and releasing f_P, a FRET-marked domain, if and only if both f_T and f_S were released. Thus both f_S and f_T have to be released (i.e both S and T are false) in order for fluorescence to occur (again, in the presence of the complementary FRET donor $\bar{f_P}$ domain.)

We note that further decoupling stacks are not necessary for combining higher-level expressions in this form: The decoupling is merely necessary to allow all

domains to interact with input domains initially, to ensure hybridization with input strands happens prior to exposing competitive gates. The result strands in higher-level expressions cannot possible react with inputs, with one exception: Use of semi- or full decoupling is again beneficial when "not"-type operators are used, as these release strands to the solution that compete with gates sharing the same sequence. If T is an expression determined by $F(T)$, $\neg T$ can be determined by releasing together with the initial stack for T, the domain f_T indicating T is false, and releasing its complement $\bar{f_T}$ in the result stack for T. Here we need decoupling to allow T to evaluate (i.e undergo reactions necessary to determine its value) prior to f_T reacting with further stacks in the expression.

We hope the reader can understand from this example the general mechanism by which we combine sub-expressions to form our final propositional logic statement P. A recursive definition of F according to the representation of the statement P as as tree, extending the definition of the initial cases above, is the following:

$$\begin{aligned} F(T \wedge S) =& F(T) \cup F(S) \cup \left\{ \dagger \bar{f_S} | f_{T \wedge S}, \dagger \bar{f_T} | f_{T \wedge S} \right\} \\ F(T \vee S) =& F(T) \cup F(S) \cup \left\{ \dagger \bar{f_S} \bar{f_T} | f_{T \wedge S} \right\} \\ F(\neg T) =& F(T) \cup \left\{ | f_{\neg T}, \dagger \bar{f_T} | \bar{f_{\neg T}} \right\} \end{aligned}$$

5.1.8. NAND gates. From the point of view of conciseness, one may have simply designed a NAND gate. NAND, standing for "NOT AND", is the boolean logic binary operator whose value is 'true' if and only if both of its operands were false. It is a basic fact of logic theory that it is universal, meaning that any of the 2^4 possible binary boolean operators can be constructed by chaining several NAND gates. Furthermore if we fix one input, we can construct the NOT logical operation. Therefore we may have simplified the definition of F somewhat, as well as made our arguments slightly shorter, without losing the generality of our results. However, as is the case with electronic circuit design, using AND, OR and NOT gates is both more efficient in terms of the number of components in the circuit, as well as being easier to reason about. The construction of the NAND gate in our case can be read off of the recursive definition of $F(\neg(a \wedge b))$ and interested readers may follow that direction.

5.1.9. Asymptotics. We now proceed to discuss some emergent properties of the model. We wish to count the number of stacks N and number of reactions R necessary to evaluate a proposition P over the domains D. First we note that the initial stacks in $F(P)$ for some P need not be counted: If $P = \ell(S,T)$ (here ℓ stands for either logical 'OR' or logical 'AND'), we can combine the initial stacks of S and T into one long stack releasing all the initial constituents. As we do this for the whole tree, we discover that in essence, we just have to include all of the domains in D in a single stack. This stack can be unwound immediately, the rest of the evolution of the computation progressing by interactions between the result strands f_{P_i}, f_{P_j} for some sub-expressions P_i, P_j of P. There is no need to introduce further initial stacks. This is evident from the definition of F above. The only exception to this is when we negate an expression, computing $\neg P_i$. Then we need to introduce one more strand, the fluorescent receiver domain f_{P_i}, into the solution initially. In the notation for F, we do this as a separate ungated stack. But this is not compulsory – rather, it simplifies the notation for F to introduce it separately. We can in fact simply include this instruction as part of the initial

stack containing all domains in D. To summarize, there need only ever be one initial stack no matter how complex the expression P is. Furthermore, the size of that initial stack is always $|D| + M_{\text{neg}}$ instructions, where M_{neg} is the total number of logical NOT operators used in the expression. In our exploration of the effect of logical tautologies on the parameters N and R we will treat the effect of different equivalent forms of P on M_{neg}.

We see that for a proposition $P = d_1 \wedge \ldots \wedge d_n$, $F(P)$ has n result stacks in half-decoupled form, each with two gates and one instruction. For $P = d_1 \vee \ldots \vee d_n$, $F(P)$ we have one result stack, gated by $n+1$ domains, and containing a single instruction. In fact, $n+1$ domains are needed only for the half-decoupled in the case that the stack is gated only by first level domains. By this we mean the domains that may appear in the input, as opposed to being gated by the result strands from stacks for other clauses. Instead of accurately capturing the different conditions it will suffice to say we require $o(n)$ gates in that case. This also holds for conjunctions and disjunctions of higher-order expressions P_i, so the same results hold for expressions of the form $F(P_1 \wedge \ldots \wedge P_n)$ and $F(P_1 \vee \ldots \vee P_n)$.

Already a curiosity arises in relation to the construction of a tree of propositions. Where the logical expression $(a \wedge b) \wedge (c \wedge d)$ is equivalent to $a \wedge b \wedge c \wedge d$ (i.e AND is associative), in our system $a \wedge b \wedge c \wedge d$ is preferable from a number of stacks point of view as it requires 4 stacks, compared to 6 required by $(a \wedge b) \wedge (c \wedge d)$. The latter has two stacks for each clause and two for the combination $S \wedge T$. On the other hand, for $d_1 \vee \ldots \vee d_n$ we need one stack with $o(n)$ gates in the un-parenthesized case, whereas each parentheses introduced, so to speak, costs us an additional stack and an additional gate. When designing large queries then, we will be guided by altering the sequence to arrive at forms that include long sequences of conjunctions or disjunctions, rather than many alternating cases. In electronic circuit design, coincidentally, placing the parentheses differently does not affect the number of components, though it does effect the length of the longest chain of operations. Similarly, we turn to the analysis of the number of reactions.

5.1.10. Longest critical chain. Specifically the interesting measure of the number of reactions is not the aggregate number but rather the longest chain of reactions that have to occur one after the other. In essence this chain is the critical path of the whole computation and determines the time required to run the computation. This is similar to the notion of T_∞ in parallel algorithm design. We ignore the time required to unwind stacks as this involves only clock reactions which are orders of magnitude faster than stack ungating and instruction hybridization reactions. We have the following properties:

- For a conjunction of n domains, all reactions are parallel. The value is determined within the duration of one reaction after all initial domains have been released. That is because each stack is gated by a single domain. In the case of semi-decoupled or fully-decoupled stacks we in fact need an extra constant number of reactions. But as before, we can simply say $o(1)$ reactions per conjunction are needed.
- For a disjunction of n domains, we can determine the output only after allowing the result stack to become ungated. As it has $o(n)$ gates, this will require $o(n)$ reactions.

- A negation requires $o(1)$ reactions: There is an additional reaction beyond the gate reaction that we must wait for – the binding of $f_{\neg a}$ to the result strand $\bar{f}_{\neg a}$, but it is still a constant number of reactions.

One immediately sees the interesting trade-off between OR and AND operations arising from our model: AND clauses are evaluated faster, but require more stacks. The requirement for more stacks is a practical design limitation since each species of stacks needs to be constructed separately. OR stacks, on the other hand, take longer to evaluate but require a single stack (though it is longer). De Morgan's law for converting $\neg a \vee \neg b$ to $\neg(a \wedge b)$ has a very concrete effect: It halves the time required at the cost of twice the number of stacks. In fact, we can see that an expression P over n domains, once transformed to conjunctive normal form, will run in at most $o(n)$ reactions even if it ends up having $2^{\frac{n}{2}}$ disjunction clauses. This might involve $o(2^{\frac{n}{2}})$ different stacks, but it is still remarkable to parallelize so trivially such that we can evaluate the expression in just $o(n)$ time.

We defer the full analysis of the optimizations that can be made to the solutions of this problem using logical tautologies to future work. There has been much work in the area of optimizing propositional logic expressions under the constraints of implementation by electronic gates and that literature can serve as a basis for anyone seeking to explore this further. We will, however, finish this section by applying the analysis to our running example of the predicate $(a \wedge b) \vee (c \wedge d)$. As it is written, it requires five stacks: Four for the conjunctions and one for the disjunction. The time we have to wait before introducing the $\bar{f}$ fluorescent probe strands in order to allow all reactions to occur is three reactions long: Two for the disjunction and one for all conjunctions. If we use half decoupled stacks this grows to four, to account for two reactions necessary to ungate the conjunction stacks. However, if we use De-Morgan's law to turn this proposition into the logically equivalent $\neg(\neg(a \wedge b) \wedge \neg(c \wedge d))$, we now use one more stack, for a total of six stacks: Two for each conjunction. The longest chain of reactions is extended by the multiple negations so we do not arrive at a shorter reaction time, and this does not gain us anything. However, if we apply De-Morgan's law once more, to each of the conjunctions, we get $\neg((\neg a \vee \neg b) \wedge (\neg c \vee \neg d))$. This now requires just four stacks (one stack less than our original expression), and the number of reactions until we can tell the result is now six: Each of the disjunctions requires four reactions to calculate the result, including the negations and half-decoupling. The negated conjunction takes two more. A final use of De Morgan's law will turn this into $(\neg(\neg a \vee \neg b) \vee \neg(\neg c \vee \neg d))$. Here only three stacks are needed (two less than the original expression) but the number of reactions is the highest at seven reactions. Interestingly, To accurately detect $(a \wedge b) \vee (c \wedge d)$ in a different system we need just four results stacks: one gated by a, c, another by b, d, another by a, d and a final one by b, c. This ensures that at least the combinations of a and b or c and d must be present in the solution to prevent any stack from opening, and any other combination of two or less of the domains a, b, c, d in the input will cause one of the stacks to open. The reaction length is two reactions, or three for the half-decoupled version. This is equivalent to the expression $(a \vee c) \wedge (b \vee d) \wedge (a \vee d) \wedge (b \vee c)$, but it omits the conjunctive stacks by simply using the same fluorescent activator strand f for all disjunction stacks, so that if any disjunction failed the entire expression fails immediately. This optimized version works in this case because of the number of domains involved and the type of expression, given in disjunctive normal form.

It saves us both stacks and reaction time only because for $n = 4$, $4 = 2^{\frac{n}{2}} = n$ so that we lose nothing by the exponential growth of stacks.

5.2. Digital Electronic Logic Circuit Analogy

In a way, there is an analogy between the solution of our so-called recognition problem and familiar logic circuits. This analogy is both interesting and will help elucidate our second example problem. Circuits implementing a specific logical expression are built in a tree with input wires $e_1, \ldots, e_n$ and an output wire f. The input wires carry voltage to signify that e_i is true, and no voltage otherwise. Gates like AND and OR combine two input wires by producing or not producing voltage on an output wire o_j leading out from them to other AND and OR gates, until a single wire f leads out of the circuit with a charge indicating the expression evaluated to true, and no charge otherwise. A NOT gate places electric charge on its output wire if its input had none (usually using an auxiliary gate, constantly charged, since current cannot be generated without a source of power). Physical location in a circuit – the ordering of components and the wiring between them – determines the combination of the inputs that produce the output. The location, which we abstractly label by $e_1, \ldots, e_n, o_1, \ldots, o_m, f$ is determined in theory by the underlying physical model (wires have to run into a gate and out of it) and then in practice using design software implementing trace and route algorithms which place the wires on a chip. In the DNA program, there is no concept of location: multitudes of molecules can interact with multitudes of other molecules in a uniformly mixed solution. However, the role of locations is fulfilled by *sequences* of bases, specifying very precisely which strand interacts with what other strand – equivalent to the wiring scheme of a circuit specifying which wire runs to which gate. These we too abstract using labels (the domains we've used so far like a, b, d_i, s, f) and their realization is handled by "low-level" software determining the exact sequence – NUPACK, managed by our compiler's understanding of the model (which is equivalent to the physical model underlying the electronic circuit). The necessity to include an extra input strand for every NOT operator in the expression is similar to the requirement in electronic circuit design for NOT gates to receive an extra input wire carrying charge. Finally, the tree structure for decomposing logical expressions, used in evaluating complicated propositions has the exact same structure in both cases. There is a converse analogy in reading the output of both systems as well. Where voltage on the output wire (existence of electron flow, essentially) is usually interpreted as the logical expression being satisfied in the case of electronics, we interpret the existence of the result strand as a false proposition, and vice-versa. The key observation in this analogy is that the specificity of current attained by wiring in electronic circuits (a spatial property of the model) is replaced by the specificity of DNA hybridization using sequences. The fact that DNA base sequences allow us to control which molecules interact with each other will instruct our intuition on how to proceed in solving further problems using the DNA computation model.

5.3. Future research directions

Many questions arise from this model, not the least of which being its experimental applicability. While experimental evidence of orderly construction and unwinding of stacks as predicted by the model has been demonstrated by Daly in

Chang and Shasha's original work [**3**], an experiment involving many gated stacks together with FRET proceeding accurately has yet to be conducted. Further theoretical questions involve the limits of this model's computability. We have not demonstrated it is Turing Universal and in fact have not even shown it to be equivalent to the class of languages represented by Finite Automata, although we suspect this to be the case. At least one constraint will have to be imposed on the equivalence, and that is having a bound on the length of input consumable by the automaton (until, in our case, it runs out of molecules able to further react). Other interesting problems may be worked out in our model. There is also further work on simulating low-population behavior in the compiler's code, or enriching it by allowing higher level languages, closer to modern procedural programming languages, to generate a description of the domains and stacks involved.

Bibliography

[1] Adleman. Molecular computation of solutions to combinatorial problems. *Science, New Series*, 266(5187):1021–1024, November 1994.

[2] Yaakov Benenson, Ehud Shapiro, and et al. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(430), November 2001.

[3] Joey Chang and Dennis Shasha. *Storing Clocked Programs inside DNA*. Synthesis Lectures on Computer Science. Morgan and Claypool, 2011.

[4] Didienko. Dna probes using flourescent resonance energy transfer design and applications. *Biotechniques*, 31(5):1106–1121, November 2001.

[5] Dirks and Pierce. Triggered amplification by hybridization chain reaction. *PNAS*, 101(43):15275–15278, October 2004.

[6] Dirks, Erik Winfree, and Pierce. Thermodynamic analysis of interacting nucleic acid strands. *SIAM Review*, 49(1):65–88, January 2007.

[7] Elisa Franco, Erik Winfree, and et al. Timing molecular motion and production with a synthetic transcriptional clock. *PNAS*, 108(40):E784–E793, October 2011.

[8] Yang Gao, Lauren K. Wolf, and Rosina M. Georgiadis. Secondary structure effects on dna hybridization kinetics: a solution versus surface comparison. *Nucleic Acids Research*, 34(11):3370–3377.

[9] Green and Tibbets. Reassociation rate limited displacement of dna strands by branch migration. *Nucleic Acids Research*, 9(8):1905–1918, 1981.

[10] Elizaveta Kossoy, Noa Lavid, Michal Soreni-Harari, Yuval Shoham, and Ehud Keinan. A programmable biomolecular computing machine with bacterial phenotype output. *ChemBioChem*, 8(11):1255–1260, 2007.

[11] Matthew R. Lakin, Simon Youssef, Filippo Polo, Stephen Emmott, and Andrew Phillips. Visual dsd: a design and analysis tool for dna strand displacement systems. *Bioinformatics*, 2011.

[12] Liekens and Fernando. Turing complete catalytic particle computers.

[13] Jurgen Pahle. Biochemical simulations stochastic approximate stochastic and hybrid approaches. *Briefings in Bioinformatics*, 10(1):53–64, January 2009.

[14] Riedel, Shea, and et al. Writing and compiling code into biochemistry.

[15] Rothemund and Adleman. Solution of a 20-variable 3-sat problem on a dna computer. *Science*, 296:499–502, April 2002.

[16] Paul Rothemund. Folding dna to create nanoscale shapes and patterns. *Nature*, 440:297–302, March 2006.

[17] Paul W. K. Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of dna sierpinski triangles. *PLoS Biology*, 2(12):e424, December 2004.

[18] SantaLucia and et al. A unified view of polymer dumbbell and oligonucleotide dna nearest-neighbor thermodynamics. *Proc. Natl. Acad. USA*, 95:1460–1465, February 1998.

[19] Ehud Shapiro. Molecular implementation of simple logic programs. *Nature Nanotechnology*, August 2009.

[20] Soloveichik, Seelig, and Winfree. Dna as a universal substrate for chemical kinetics. *PNAS early edition*, January 2010.

[21] Stoelinga. An introduction to probabilistic automata.

[22] Wang and Drlica. Modeling hybridization kinetics. *Mathematical Biosciences*, 183:37–47, 2003.

[23] Watson, Camien, and Warner. Kinetics of branch migrationin double-stranded dna. *Proc. Natl. Acad. Sci. USA*, 73(7):2299–2303, July 1977.

[24] Winfree, Cook, and Sloveichik. Programmability of chemical reaction networks.

[25] Winfree and et al. Bistability of an in vitro synthetic autoregulatory switch. *Arxiv*, January 2011.

[26] Zadeh, Wolfe, and Pierce. Nucleic acid sequence design via efficient ensemble defect optimization. *Journal of Computational Chemistry*, 32(3), 2010.

[27] Zhang and Erik Winfree. Control of dna strand displacement kinetics using toehold exchange. *Journal of American Chemical Society*, 131:17303–17314, November 2009.