WRITING AND COMPILING CODE INTO BIOCHEMISTRY*

ADAM SHEA, BRIAN FETT, MARC D. RIEDEL and KESHAB PARHI
Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, Minnesota 55455
Email: {shea0097, fett, mriedel, parhi}@umn.edu

This paper presents a methodology for translating iterative arithmetic computation, specified as high-level programming constructs, into biochemical reactions. From an input/output specification, we generate biochemical reactions that produce output quantities of proteins as a function of input quantities performing operations such as addition, subtraction, and scalar multiplication. Iterative constructs such as "while" loops and "for" loops are implemented by transferring quantities between protein types, based on a clocking mechanism. Synthesis first is performed at a conceptual level, in terms of abstract biochemical reactions – a task analogous to *high-level program* compilation. Then the results are mapped onto specific biochemical reactions selected from libraries – a task analogous to *machine language* compilation. We demonstrate our approach through the compilation of a variety of standard iterative functions: multiplication, exponentiation, discrete logarithms, raising to a power, and linear transforms on time series. The designs are validated through transient stochastic simulation of the chemical kinetics. We are exploring DNA-based computation via strand displacement as a possible experimental chassis.

1. Introduction

Recent accomplishments in synthetic biology portend of a coming revolution. From *Salmonella* that secretes spider silk proteins,[1] to yeast that degrades biomass into ethanol,[2] to *E. coli* that produces antimalarial drugs,[3] the potential impacts are far-reaching.

The scope of the field is, in fact, broader. The J. Craig Venter Institute's team has made significant progress toward the goal of artificial life: a living bacterial cell with fully synthetic DNA.[4,5] In engineering terms, the objective is to assemble a machine (a synthetic bacterium) in which the functionality of all the parts (the genes, the proteins that they code for, and how these interact biochemically) are understood. If the machine works, this vindicates the scientific understanding; if it doesn't – and surely it won't at first – then new understanding can be achieved by examining where and how it breaks. Of course, with a working blueprint for a synthetic machine, new functionality can be engineered robustly and effectively.

The set of constitutive parts that can be used for genetic manipulation in synthetic systems is vast. Comprehensive repositories of genetic data have been assembled – some public, some commercial – cataloging genes, their DNA sequences, and their products. A concerted effort has been made to assemble repositories of standardized and interoperable parts for synthetic applications. The platforms used will depend on the application, but the technology for synthesizing DNA is becoming routine: firms have started offering custom-gene synthesis through e-commerce websites (the going rate is $0.49 per base pair). So, in a real sense, the hardware for synthetic biology exists, i.e., the technology and infrastructure for obtaining cells with custom-designed genes. The instruction set is, to a large extent, known, i.e., genes and their function, cataloged in libraries. The challenge is: *how can we write code with these instructions on this type of hardware*?

Conceptually, the rules of biochemistry are straight-forward: each biochemical reaction is a primitive process that specifies how and at what rate different types of proteins combine to form other types of proteins. The complexity stems from the dynamics at play among the multitude of coupled reactions operating on the different protein types, asynchronously and in parallel. Techniques for *analyzing* such processes are well established.[6] However, *synthesizing computation* with such mechanisms requires entirely new techniques – and an entirely new mindset.

One of the great successes of computer engineering has been in abstracting and scaling the design problem. The physical behavior of transistors is understood in terms of differential equations – say, with models found in tools such as SPICE.[7] However, the design of circuits proceeds at a more abstract level – in terms of switches, gates, and functional units. Software is conceived of and validated independently of the hardware platform. This modular approach makes the design tractable; furthermore, it permits a systematic

exploration of different configurations, leading to optimal designs. Although driven by experimental expertise, synthetic biology has reached a stage where it calls for a similar degree of modularization and abstraction.

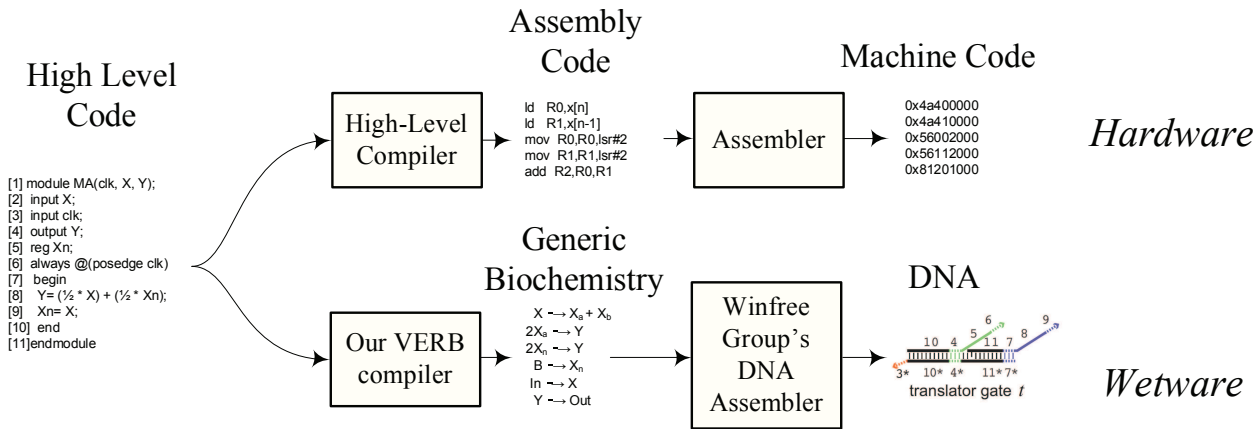

Fig. 1: Analogy between compiling code for hardware and for biochemistry.

2. Overview

Our approach brings a design perspective: we tackle the problem of synthesizing biochemical reactions that implement specified input/output functionality. From a specification in a high-level language like C or Pascal, we generate biochemical reactions that produce output quantities of proteins as a function of input quantities, performing operations such as addition, subtraction, and scalar multiplication. Iterative constructs such as "while" loops and "for" loops are implemented by transferring quantities between protein types, based on a clocking mechanism. We demonstrate our approach through the compilation of a variety of standard iterative functions: multiplication, exponentiation, discrete logarithms, raising to a power, and linear transforms on time series. The designs are validated through transient stochastic simulation of the chemical kinetics.

2.1. *Computation with Biochemistry*

Interesting biochemistry typically involves complex molecules such as proteins and enzymes. Within the confines of a cell, the *quantities* of such molecules are often surprisingly small: on the order of tens, hundreds, or thousands of molecules of each type. At this scale, individual reactions matter and the problem must be modeled discretely.[6]

In our view of biochemical computation, the quantities of proteins are whole numbers (i.e., non-negative integers). We will refer to these quantities as "*registers*". Biochemical reactions alter these quantities: the reactions fire repeatedly, modifying the protein quantities by small integer amounts. Consider the reaction

$$a + b \xrightarrow{\text{fast}} 2c. \tag{1}$$

When this reaction fires, one molecule of a is consumed, one of b is consumed, and two of c are produced. (Accordingly, a and b are called the *reactants* and c the *product*.) Each reaction has an associated *rate* (listed above the arrow in our notation). Given several reactions, the probability of each firing is proportional both to its rate and to the quantities of its reactants present. Although we refer to rates in relative and qualitative terms – e.g., "fast" vs. "slow" – these are, in fact, quantitative values that are either deduced from biochemical principles or measured experimentally. The functionality of a biochemical system can be analyzed using stochastic simulation.[6,8,9]

Our contribution is to tackle the problem of computation at this abstract level – working not with specific molecular types but rather with arbitrary types (a, b, c, etc.). This is illustrated in Figure 1. For conventional hardware, programs are specified in a high-level language like C; a compiler translates this into assembly

language; then an assembler produces the machine code. In our bio-design flow, we begin with the same sort of high-level description. (We use Verilog, a hardware description language.[10]) Our prototype compiler called VERB (Verilog Elements for Register-Based Biochemistry) compiles these specification into generic biochemical reactions. Then this design is mapped on a chemical substrate. The end result is a description of the actual biochemistry: protein-protein reactions or DNA interactions.

2.2. *Compiling the Programs into Biochemistry*

A possible experimental chassis for our method is the mechanism of DNA-based computation advocated by Erik Winfree's group at Caltech.[11] They have shown that the kinetics of arbitrary chemical reactions can be implemented through DNA strand-displacement reactions. They provide an assembler that accepts a set of biochemical reactions with nearly any rate structure and delivers the corresponding DNA sequences for the displacement reactions. Reaction rates are controlled by designing sequences with different binding strengths; the binding strengths are controlled by the length and sequence composition of toeholds.[11] Our contribution can be positioned as the "front end" of the compilation flow; the DNA assembler and experimental chassis described by these authors constitute the "back-end."

3. Related Work and Context

There has been considerable research directed at the question of computation with genetic regulatory mechanisms.[12] DNA and RNA-based computation have been explored theoretically and demonstrated experimentally.[13–15] Mathematical expertise from control and dynamical systems has been applied to the analysis of biochemical systems.[16] Oscillatory mechanisms, suitable for the sort of clocking used in our designs, have been demonstrated experimentally.[17] Samoilov, Arkin and Ross established a comprehensive analytic framework for studying the dynamics of biological systems in terms of the signal processing functions that they perform.[18] Soloveichik, Cook, Winfree and Bruck discuss theoretical aspects of molecular computation.[19] The concepts of register-based computation and clocking that we use are due to these authors.[19]

4. A Toolkit for Biochemical Arithmetic

We describe elements of flexible toolkit of functional modules. In our view of biochemical computation, the input quantities of proteins are non-negative integers. Computation is implemented by biochemical reactions. These fire repeatedly, modifying the protein quantities by small integer amounts; the end results are output quantities of proteins. The challenge in setting up such computing, of course, is that the biochemical reactions execute asynchronously and in parallel.[20]

The computation that we propose in the modules below is exact and independent of the specific values of the rates, although it requires that the rates in different categories differ by a sufficient amount. For instance, we assume that when a "fast" reaction can fire it does so – repeatedly, until it runs out of reactants – before a "slow" reaction ever fires.

4.1. *Addition and Scalar Multiplication*

The first and simplest of these computations is addition with scalar multiplication. We can implement

Scalar Multiplication and Addition

$$\boxed{|z| = \tfrac{a}{b}|x| + \tfrac{c}{d}|y|}$$

Reactions:

$$bx \longrightarrow az$$
$$dy \longrightarrow cz$$

Fig. 2: **A biochemical module for addition and scalar multiplication.** When the first reaction fires, it consumes b molecules of x and produces a molecules of z. When the second reaction fires, it consumes d molecules of y and produces c molecules of z. When both reactions have fired to completion, the number of molecules of z will be the scaled sum of the number molecules x plus the number of molecules of y.

this by merely choosing reactions with the correct stoichiometry. This is shown in Figure 2. (Here and throughout we use $|\cdot|$ to denote the quantity of a type of molecule.)

4.2. *Multiplication*

A biochemical construct for performing multiplication is shown in Figure 3. In this set of reactions, note that none can fire until the first one does, producing a molecule of type i. When it does, it initiates an iteration of a *loop*: the quantity of z increases as the second reaction fires repeatedly until there is no more y remaining. Once this process terminates, the third and fourth reactions fire, ending the iteration and restoring y to its initial value. In each iteration, the quantity of x is decremented by one and the quantity of z is incremented by y. The final result is a quantity of z equal to the initial quantity of x times the quantity of y.

4.3. *More Complex Arithmetic*

Modules for computing exponentiation, discrete logarithms and raising to a power, are shown in Figures 4, 5, and 6, respectively. With the latter, our scheme can be used to implement arbitrary polynomial functions; hence, in principle, it could be used to approximate complex functions through Taylor series expansions.

Multiplication

$$\boxed{|z| = |x| \times |y|}$$

Pseudo-code:

```
while x > 0 {
   z = z + y
   x = x - 1
}
```

Reactions:

$$\begin{aligned} x &\xrightarrow{\text{slowest}} i \\ i + y &\xrightarrow{\text{fastest}} i + y\prime + z \\ i &\xrightarrow{\text{fast}} \varnothing \\ y\prime &\xrightarrow{\text{slow}} y \end{aligned}$$

Fig. 3: **A biochemical module for multiplication.** The module consumes molecules of x one at a time, adding the quantity of y to the quantity of z each time. (Here i and $y\prime$ are intermediate types; it is assumed that no molecules of these types are present initially. The symbol $\varnothing$ as a product indicates "nothing," meaning that the type degrades into products that are no longer tracked or used.)

Exponentiation

$$\boxed{|y| = 2^{|x|}}$$

Pseudo-code:

```
y = 1
while x > 0 {
  y = 2 * y
  x = x - 1
}
```

Reactions:

$$\begin{aligned} c + 2y &\xrightarrow{\text{fast}} c + y \\ c &\xrightarrow{\text{medium}} \varnothing \\ x &\xrightarrow{\text{slow}} i \\ i + y &\xrightarrow{\text{faster}} i + 2y\prime \\ i &\xrightarrow{\text{fast}} \varnothing \\ y\prime &\xrightarrow{\text{medium}} y \end{aligned}$$

Fig. 4: **A biochemical module for exponentiation.** First, a pair of reactions set the quantity of y to 1. Then molecules of x are consumed one at a time, doubling the quantity of y each time. (Here c and $y\prime$ are additional types; it is assumed that initially there is some non-zero quantity of c, and zero quantity of $y\prime$.)

Logarithm

$$\boxed{|y| = log_2(|x|)}$$

Pseudo-code:

```
while x > 1 {
  x = x/2
  y = y + 1
}
```

Reactions:

$$\begin{aligned} b &\xrightarrow{\text{slow}} a + b \\ a + 2x &\xrightarrow{\text{faster}} c + x' + a \\ 2c &\xrightarrow{\text{faster}} c \\ a &\xrightarrow{\text{fast}} \varnothing \\ x' &\xrightarrow{\text{medium}} x \\ c &\xrightarrow{\text{medium}} y \end{aligned}$$

Fig. 5: **A biochemical module for computing a base-2 logarithm.** The input x repeatedly halves itself; each time it does so, y is incremented by one. (Here a, b, c and x' are are additional types; it is assumed that initially there is some non-zero quantity of b, and zero quantity of a, c and x'.)

Raising to a Power

$$\boxed{|y| = |x|^{|p|}}$$

Pseudo-code:

```
y = 1
d = 0
while p > 0 {
  w = x
  while w > 0 {
    d = d + y
    w = w - 1
  }
  y = d
  d = 0
  p = p - 1
}
```

Reactions:

$$\begin{aligned} c + 2y &\xrightarrow{\text{fast}} c + y \\ c &\xrightarrow{\text{medium}} \varnothing \\ p &\xrightarrow{\text{slowest}} a \\ a + x &\xrightarrow{\text{medium}} b + a + x' \\ b + y &\xrightarrow{\text{fastest}} y' + d + b \\ b &\xrightarrow{\text{faster}} \varnothing \\ y' &\xrightarrow{\text{fast}} y \\ a &\xrightarrow{\text{slow}} e \\ e + y &\xrightarrow{\text{faster}} e \\ e + x' &\xrightarrow{\text{faster}} e + x \\ e &\xrightarrow{\text{fast}} \varnothing \\ d &\xrightarrow{\text{slower}} y \end{aligned}$$

Fig. 6: **A biochemical module for computing raising to a power.** First, a pair of reactions set the quantity of y to 1. Then a pair of nested loops achieves the computation: the inner loop computes $|x|$ times the current result (starting with 1); the outer loop does this operation $|p|$ times. (Here a, b, c, d, e, x', and y' are are additional types. It is assumed that that quantities of all of these except for c are initially zero; the initial quantity of c is any non-zero value.)

4.4. *Clocking*

An important constraint in our design methodology is the timing captured in the relative rates of the biochemical reactions. With modules such as multiplication described above, there is an implicit ordering of the reactions. To achieve this, the reaction rates must sometimes be separated by orders of magnitude: some much faster than others, some much slower. This may be unrealistic.

To overcome this issue, we have have proposed a technique that we call "module-locking".[21] The scheme involves adding a *key* requirement to each phase of the computation. *Keysmiths* are produced occasionally; if other keys are present, they quickly disappear – before they can produce their key. Only if no other keys are present will they produce their key. This ensures that at most one type of key is present (thus allowing only one part of the loop to fire at a time); also it ensures that only one key of that type is present (thus allowing for re-locking). The template for reactions with this functionality is:

$$\begin{aligned} \forall i: \quad & \varnothing \xrightarrow{\text{slow}} \text{keysmith}_i \\ \forall i: \quad & \text{keysmith}_i \xrightarrow{\text{slow}} \text{key}_i \\ \forall i: \quad & \text{key}_i \xrightarrow{\text{slow}} \varnothing \\ \forall i,j: \quad & \text{key}_i + \text{keysmith}_j \xrightarrow{\text{fast}} \text{key}_i \end{aligned} \tag{2}$$

This is for all i, j phases of the computation, e.g., steps in an operation like multiplication. (The symbol $\varnothing$ as a reactant indicates that the reaction does not alter the quantity of the reactant types, perhaps because the quantity of these is large or replenishable; in such cases we can assume that the quantity is simply unity and adjust the rate accordingly). The first reaction of each phase must be modified so that it depends on the key:

$$\text{key}_i + \text{reactants} \rightarrow \text{products} + \text{key}_i. \tag{3}$$

Typically, the key will be a catalyst, appearing as both a reactant and a product, but this need not be the case. With locking, our method synthesizes robust computation that is nearly rate independent, requiring at most two speeds ("fast" and "slow"). The trade-off is with respect to the size of the solution: more reactions are needed. Further details are given in our paper on the topic.[21]

5. Compiling Iterative Code

By combining arithmetic operations and our clocking mechanism, we can implement iterative operations such as linear transforms on time series. Such operations are useful for performing filtering operations.

Example 1
Consider an application that calls for biochemistry that performs a filtering operation such as computing a moving average. *Given a noisy input signal $X[n]$, a moving average filter produces an output signal $Y[n]$ that is a smoother version of the input. The function is*

$$Y[n] = \frac{1}{2}X[n] + \frac{1}{2}X[n-1].$$

where the n-th value is the current value and the $(n-1)$-st value is the previous value of each signal. The iterative operation can be specified as follows (we use the syntax of Verilog[10]):

```
[1]  module MA(clk, X, Y);
[2]    input X;
[3]    input clk;
[4]    output Y;
[5]    reg Xn;
[6]    always
[7]      begin
```

```
[8]          Y= (1/2 * X) + (1/2 * Xn);
[9]          Xn= X;
[10]      end
[11]  endmodule
```

We translate this specification into a set of biochemical reactions, as follows. Each operation is translated into a biochemical reaction with the protein types that correspond to the variables. All these reactions are keyed, according to clock phases. These reactions are show in Figure 7. For simplicity, here we omit the details of how the keys are generated. Key-keysmith reactions of the form of Equation 2 should be included for $\text{key}_0, \text{key}_1$, and key_2.

R1:	$\text{key}_0 + X \xrightarrow{\text{fast}} X_a + X_b + \text{key}_0$
R2:	$\text{key}_0 + Y \xrightarrow{\text{fast}} \text{key}_0$
R3:	$\text{key}_1 + 2X_n \xrightarrow{\text{fast}} Y + \text{key}_1$
R4:	$\text{key}_2 + 2X_b \xrightarrow{\text{fast}} Y + \text{key}_2$
R5:	$\text{key}_2 + X_a \xrightarrow{\text{fast}} X_n + \text{key}_2$
CR1:	$\text{key}_2 \xrightarrow{\text{slow}} \text{keysmith}_0$
CR2:	$\text{keysmith}_0 \xrightarrow{\text{slow}} \text{key}_0$
CR3:	$\text{key}_0 \xrightarrow{\text{slow}} \text{keysmith}_1$
CR4:	$\text{keysmith}_1 \xrightarrow{\text{slow}} \text{key}_1$
CR5:	$\text{key}_1 \xrightarrow{\text{slow}} \text{keysmith}_2$
CR6:	$\text{keysmith}_2 \xrightarrow{\text{slow}} \text{key}_2$

Fig. 7: Biochemistry implementing the moving-average filter.

We simulate and validate our designs with transient stochastic simulation.[9] The simulation results shown in Figure 8, illustrate the functionality of the design: the moving average smooths high-frequency noise. Here, the input X is shown in green; it is a noisy sinusoid. The output Y is shown in red; note that it is a clearer sinusoidal waveform.

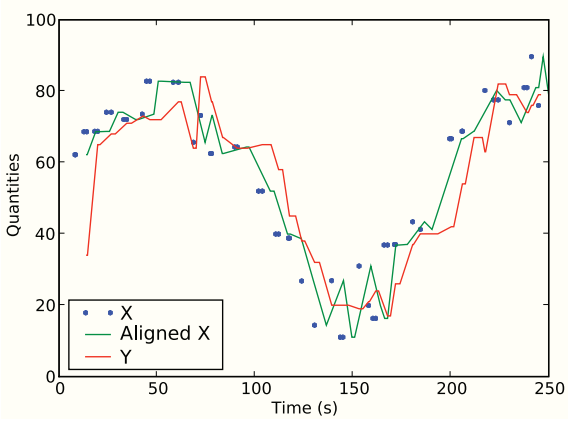

Fig. 8: Input and output waveforms for a biochemical moving-average filter. The input quantity X is shown both as points from the simulation and as the ideal curve. The output Y is shown in red. The green curve shows where the value of X is after the average delay of the system in order to be time-aligned with the red output curve.

Table 1: Compilation Results

	deserializer	vector matrix multiplier	integrator	differentiator
Reactions	27	28	14	19
Registers	23	18	4	10
Clocks	2	5	3	4

6. Additional Design Examples

We briefly discuss a few other designs that we synthesized with our compiler. All perform iterative arithmetic on non-negative integer values; these integer values correspond to time-varying quantities of input and output proteins.

- The **deserializer** is simply an 8-element shift register. Instead of operating on bits, it operates on integer values. This module is useful as a starting point for convolution-based algorithms and in signal decoding.
- The **vector-matrix** multiplier multiplies a 3-element input vector by a fixed 3x3 matrix to obtain an output vector. It can be trivially expanded to higher dimensions; the only limitation is that it has nonnegative integers for input and output.
- The **integrator** implements a "bipolar" encoding: its inputs and outputs are represented as as the difference of two protein quantities. While each of these quantities is a nonnegative integer, their difference can be any integer, positive or negative. It computes a running sum of its input values. In order to ensure that the output quantities do not increase without bound, an "equalize" operation is implemented. This reduces both the positive and negative output quantities by the same amount until one is zero.
- The **differentiator** operates with this same bipolar encoding. It computes the running difference of the last two values.

The parameters of the biochemical designs that our compiler produces are shown in Table 1: the number of reactions, the number of registers and the number of clock phases, for each.

7. Discussion

The concept of chemical reaction networks as a *program formalism* has been discussed in the literature.[19,22,23] Prior work established the formalism from a theoretical perspective. For instance, it has been shown that chemical reaction networks are computationally universal – that is to say, a chemical reaction network can be found to compute every function.[24] Interesting examples have been shown.[14,24]

And yet, to our knowledge, we are the first to consider the formalism from a design perspective. We advocate a modular and automated flow for synthesizing computation with biochemistry. The computation is specified in terms the requisite input/output behavior. It is compiled into an abstract chemical reaction network – a task analogous to "high-level" program compilation. Then the results are mapped onto specific biochemical components, selected from libraries – a task analogous to "machine-level" program compilation.

Our designs are robust to perturbations: molecules can degrade and the quantities can fluctuate; the result of the computation will still be correct. Furthermore, our designs are rate-independent: although we require that some reactions be faster than others, the result of the computation never depends on the actual rates (i.e., k, the rate of a reaction, never appears in the functions that are computed).

Although conceptual for the time being, our method has potential applications in domains of synthetic biology such as biochemical sensing and drug delivery. By deliberately applying design methodologies, one could engineer computational control over biological processes, designing reactions that produce specific outputs in response to different combinations of inputs. For instance, *decision feedback equalization* could be implemented for drug deliver applications. Filtering operations could be performed. The inputs would be

time-varying quantities of proteins; the outputs would be *high-pass*, *low-pass* or *band-pass* functions of the frequency of the changes in these quantities.

References

1. D. Widmaier, D. Tullman-Ercek, E. Mirsky, R. Hill, S. Govindarajan, J. Minshull, and C. Voigt, "Engineering the salmonella type iii secretion system to export spider silk monomers," *Molecular Systems Biology*, vol. 5, September 2009. [Online]. Available: http://dx.doi.org/10.1038/msb.2009.62
2. M. Sedlak and N. Ho, "Production of ethanol from cellulosic biomass hydrolysates using genetically engineered Saccharomyces yeast capable of cofermenting glucose and xylose," *Applied biochemistry and biotechnology*, vol. 114, no. 1, pp. 403–416, 2004.
3. D. Ro, E. Paradise, M. Ouellet, K. Fisher, K. Newman, J. Ndungu, K. Ho, R. Eachus, T. Ham, and J. Kirby, "Production of the antimalarial drug precursor artemisinic acid in engineered yeast," *Nature*, vol. 440, pp. 940–943, 2006.
4. D. Gibson, G. Benders, C. Andrews-Pfannkoch, E. Denisova, H. Baden-Tillson, J. Zaveri, T. Stockwell, A. Brownley, D. Thomas, and M. Algire, "Complete chemical synthesis, assembly, and cloning of a Mycoplasma genitalium genome," *Science's STKE*, vol. 319, no. 5867, p. 1215, 2008.
5. J. Glass, N. Assad-Garcia, N. Alperovich, S. Yooseph, M. Lewis, M. Maruf, C. Hutchison III, H. Smith, and J. Venter, "Essential genes of a minimal bacterium," *Proceedings of the National Academy of Sciences*, vol. 103, no. 2, pp. 425–430, 2006.
6. D. Gillespie, "Exact Stochastic Simulation of Coupled Chemical Reactions," *The Journal of Physical Chemistry*, vol. 81, no. 25, pp. 2340–2361, 1977.
7. L. Nagel and D. Pederson, "Simulation program with integrated circuit emphasis," *Midwest Symposium on Circuit Theory*, 1973.
8. M. Gibson and J. Bruck, "Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels," *Journal of Physical Chemistry A*, vol. 104, no. 9, pp. 1876–1889, 2000.
9. B. Chen and M. Riedel, "Stochastic transient analysis of biochemical reactions," *Pacific Biocomputing Symposium*, 2009.
10. Verilog, "IEEE Standard Verilog Hardware Description Language," *IEEE Standard 1364-2001*, 2001.
11. D. Soloveichik, G. Seelig, and E. Winfree, "DNA as a Universal Substrate for Chemical Kinetics," *International Meeting on DNA Computing*, 2008.
12. R. Weiss, S. Basu, S. Hooshangi, A. Kalmbach, D. Karig, R. Mehreja and I. Netravali, "Genetic circuit building blocks for cellular computation, communications, and signal processing," *Natural Computing*, pp. 47–84, 2003.
13. L. Adleman, "Molecular computation of solutions to combinatorial problems," *Science*, no. 11, pp. 1021–1024, 1994.
14. Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, and E. Shapiro, "An Autonomous Molecular Computer for Logical Control of Gene Expression," *Nature*, vol. 429, no. 6990, pp. 423–429, 2004.
15. M. Win and C. Smolke, "From the Cover: A modular and extensible RNA-based gene-regulatory platform for engineering cellular function," *Proceedings of the National Academy of Sciences*, vol. 104, no. 36, p. 14283, 2007.
16. H. El-Samad, H. Kurata, J. Doyle, C. Gross, and M. Khammash, "Surviving heat shock: Control strategies for robustness and performance," *Proceedings of the National Academy of Sciences*, pp. 2736–2741, 2005.
17. M. Elowitz and S. Leibler, "A Synthetic Oscillatory Network of Transcriptional Regulators," *Nature*, vol. 403, no. 6767, pp. 335–338, 2000.
18. M. Samoilov, A. Arkin, and J. Ross, "Signal Processing by Simple Chemical Systems," *Journal of Physical Chemistry A*, vol. 106, no. 43, pp. 10 205–10 221, 2002.
19. D. Soloveichik, M. Cook, E. Winfree, and J. Bruck, "Computation with Finite Stochastic Chemical Reaction Networks," *Natural Computing*, pp. 615–633, 2008.
20. B. Fett, J. Bruck, and M. Riedel, "Synthesizing Stochasticity in Biochemical Systems," *Design Automation Conference*, pp. 640–645, 2007.
21. B. Fett and M. Riedel, "Module Locking in Biochemical Synthesis," *International Conference on Computer-Aided Design*, 2008.
22. L. Cardelli, "From processes to odes by chemistry," *Springer*, p. 261, 2008.
23. P. Erdi and J. Toth, "Mathematical models of chemical reactions: Theory and applications of deterministic and stochastic models," *Manchester Univesity Press*, 1989.
24. M. Cook, D. Soloveichik, E. Winfree, and J. Bruck, "Programmability of Chemical Reaction Networks."