Synthesizing Stochasticity in Biochemical Systems

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Brian David Fett

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR A MASTER OF SCIENCE DEGREE

Marc Riedel, Adviser

June 2010

Dedication

To my loving wife, without whose assistance and insistence this would never have gotten finished.

Abstract

The science of Biology is evolving from a science of words and pictures to a science of hard numbers and equations. Biological systems are modeled mathematically, and computers are used to compute how systems will evolve using either differential equations or stochastic techniques. These techniques help scientists to understand the complex systems of interaction that Biology encompasses. The problem of designing such systems, however, is often relegated to simply finding similar behavior in a system and grafting that element into a new system.

We, however, propose a method for designing biochemical pathways in organisms according to arbitrary design. Specifically, we examine the problem of designing a system that makes a stochastic response to some input stimuli. The method allows the designer to implement a wide variety of responses to various environments.

We verify our designs, modeling the cell stochastically, using Gillespie simulations. We assume that the environment can be inferred by the chemistry within a cell and that the time-scales we operate on are short enough that the effect of diffusion across the membrane is negligible. Our designs consist of theoretical sets of reactions that if given suitable initial conditions will perform calculation within the system as well as make stochastic choices according to probabilities defined by the environment and the design.

Our method employs a modular design methodology that allows us to create functional modules working at the reaction level and create larger systems by composing those modules together. We create modules and methods for choosing between multiple outcomes as well as performing addition, subtraction, multiplication, logarithm, and exponentiation. This design system could be the first step to creating biochemical systems for carrying out arbitrary designs.

Contents

List of Tables

List of Figures

1 Introduction

Increasingly, biology is becoming a *computational* science as modeling and simulation are applied alongside experimental work in the lab [3]. Furthermore, with the advent of techniques for synthesizing and manipulating genetic material, it is striving to become an *engineering* discipline. In the nascent field of *synthetic biology*, researchers aim to create entirely new biological functions by modifying and integrating biological components in a systematic way [4]. The potential impacts are far-reaching. Recent feats of synthetic biology include cellulosic ethanol [5], anti-malarial drugs [6], and tumor detection [7].

By custom-designing the genetic material of organisms, such as yeast and *E. coli*, it is possible to directly synthesize biochemistry for applications. In principle, the approach could produce biochemical reactions of nearly any form. However, designing a set of reactions to implement a desired functionality – efficiently and robustly – is a challenging problem.

Biochemical systems are typically characterized through computationally intensive Monte Carlo simulations [8, 9]. More recently, some have studied biochemical reactions from a theoretical perspective, for instance proving universality [10], while others have discussed the implementation of signal processing functions [11].

Here we propose a framework for designing biochemical reactions at a theoretical level. That is, we discuss chemical species in terms of abstract types (labeled a, b, *etc.*), rather than actual biochemical types. The process of mapping these reactions to such actual chemical species is work we leave to others. We propose ways to create biochemical pathways that behave in both well defined deterministic fashions in addition to well defined stochastic fashions. We also show an alternative construction that allows us to trade off reaction rate requirements with the required reaction number while preserving the accuracy of the computations. These three quantities can be traded off in various ways allowing for the designer to decide which to preserve and which to sacrifice.

1.1 Reaction Syntax

In this paper, we will represent a reaction as follows:

$$a + 2b \xrightarrow{k} c + 3d. \tag{1}$$

The two chemical types on the left (a and b) are called reactants and are used up in the reaction (the coefficient 2 shows that two molecules of type b are used). The types to

the right(c and d) are called products because they are created by the reaction. The rate k denotes the relative rate of the reaction; when more than one reaction is considered, reactions with a higher rate will occur more rapidly. We will often use relative terms like "fast" and "slow" in place of actual numbers in this paper. These relative speeds: slowest, slower, slow, medium, fast, faster and fastest, are ranked in order of increasing value of k.

1.2 Stochastic Modeling

We will use Gillespie's method[8] for analyzing biochemical systems. In this model, we consider the state of the system to consist of non-negative integer quantities of all modeled chemical types. A single reaction is then picked out of the set of all considered reactions. The reaction updates the state, and we then pick another reaction at random. Reactions are picked until the state enters some region of the state space that we consider an end condition. These regions include states where no reaction is possible or regions where the system is known to behave in a certain way from that point onward.

The process of picking a reaction from any given state requires calculating the propensities of all reactions. Given the current quantities of the various chemical types ($|a|$, $|b|$, *etc.*) for that state. The *propensity* of any given reaction is the product of the reaction's rate and the number of combinations there are for choosing its reactants. That is to say:

$$P_i = k_i \binom{|a|}{c_a} \binom{|b|}{c_b} \cdots \tag{2}$$

where P_i is the propensity of the reaction, k_i is the reaction's rate, c_a and c_b are the coefficient of a and b (respective) on the reactant side of the reaction, and $\binom{n}{r}$ denotes the number of ways that we can choose r samples out of a population of size n.

The probability (p_i) of each reaction is then simply its propensity normalized by the sum of all propensities.

$$p_i = \frac{P_i}{\sum_j P_j} \tag{3}$$

1.3 Example Case: Multiplication

In this section, we provide a simple example of a synthesized reaction set where the number of molecules of two chemical types within a closed system are multiplied to yield a quantity of a third:

$$|z| = |x| \cdot |y|.$$

The following set of reactions accomplishes this [1]:

$$\begin{aligned} x &\xrightarrow{\text{slowest}} a \\ a + y &\xrightarrow{\text{fastest}} a + y' + z \\ a &\xrightarrow{\text{fast}} \varnothing \\ y' &\xrightarrow{\text{slow}} y. \end{aligned} \tag{4}$$

Since the rates are relative, the fastest reaction that can fire is assumed to do so – repeatedly, until it runs out of reactants – before a slower reaction ever fires. Depending on the quantities involved and the accuracy that is called for, an order of magnitude difference between two rate categories might suffice; typically however, it would be several orders of magnitude. Here a and y' 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 chemical types produced are no longer tracked, either because they are not used by subsequent reactions, or they are so common that the effects of the reaction on their quantity can be ignored.

To see that these reactions implement multiplication, first note that no reaction can fire before the first reaction produces a molecule of type a. When it does, it initiates an iteration of a *loop*: the quantity of z increases as the second reaction, being the fastest reaction, fires repeatedly until there is no more y remaining. Once this process terminates, the third and then 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 initial quantity of y.

2 Modular Synthesis Scheme

Our synthesis scheme is designed to be *modular*: using relatively small *modules* each with a very specific purpose. These modules can then be chained together — with the products of one set of reactions feeding the next — to perform operations of arbitrary complexity. The goals of this design are two-fold. First, simplicity: the user need not understand why a module works, just the input/output behavior of each. Second, extensibility: new modules can be created relatively easily, and existing modules can be replaced with better performing ones.

2.1 Deterministic Module: quantities to quantities

The deterministic module allows for computation to take place. It will take, as inputs, quantities of certain chemical types and will yield, as outputs, quantities of different types. The quantities of the output types will be well defined functions of the input quantities.

By combining these functional sub-modules, we can create biochemical calculators with output chemical quantities being any arbitrary function of an input chemical quantity.

Example 1 Simple Deterministic System

Suppose that we wanted output quantities of y and z to be dependent on the input quantity of x. Specifically, we wanted these quantities to fit:

$$|y| = 10 + 2|x| \tag{5}$$

$$|z| = 50 - |x|. \tag{6}$$

The constant term of the equations simply requires the relevant quantities to be set at these amounts at the start, however the variable term requires x to modify these initial quantities.

To accomplish this, we need each molecule of x to affect both y and z. This can be done by creating a 'custom' reaction or reaction set, such as

$$z + x \rightarrow 2y, \tag{7}$$

or by combining a few pre-made reaction sets that accomplish a known objective: in this case, fan-out, linear scaling and subtraction respectively.

$$x \rightarrow x_1 + x_2 \tag{8}$$

$$x_1 \rightarrow 2y \tag{9}$$

$$x_2 + z \rightarrow \varnothing \tag{10}$$

The advantage of the first method is that it can create very efficient and concise systems (in our example, a single reaction), however, it requires an intimate knowledge of the way the system behaves and is difficult to scale. The advantage of the second method is that people with without any such intimate knowledge can employ a basic knowledge of the relevant modules and arrive at the correct solution (additionally, the second solution computes $|y|$ correctly even if there is insufficient z). This is analogous to between programming directly in assembly versus using a programming language with a compiler, or designing circuits

with amplifiers instead of transistors. □

2.1.1 Functional Modules

Here, we define a set of functional modules that, when fitted together, can produce most desired functional dependencies[1].

In describing the functions that the modules implement, we add subscripts to the quantities of molecular types to denote *when* these quantities exist: zero indicates that this is the initial quantity, whereas infinity indicates that it is the quantity after the module has finished.

We also use the notation $\varnothing$ to denote types whose quantity we wont track. As a product this is any type that is not used as a reactant later. As a reactant, it can be any type whose quantity is not affected by the reaction (*e.g.* DNA is not used up when transcribing proteins). In either case, it could be a type whose quantity is sufficiently large that the affect of adding or removing some of it is negligible.

We have designed Subtraction, Fan-out, Linear, Multiplication, Exponentiation, Logarithmic, Power and Isolation modules. Addition is not listed as a module because it is trivial to have two modules produce the same output to achieve an additive effect.

Subtraction:

$$\boxed{|y|_\infty = |y|_0 - |x|_0}$$

Despite addition not meriting its own module, subtraction does because we are working with non-negative quantities. Therefore, subtraction cannot be done by simply adding a negative quantity. By destroying a molecule of our "output" type for each available input, we accomplish the same effect:

$$y + x \rightarrow \varnothing. \tag{11}$$

Fan-out:

$$\boxed{|y_1|_\infty = |x|_0, |y_2|_\infty = |x|_0, \ldots, |y_n|_\infty = |x|_0}$$

[1]Some functions, such as trigonometric functions, do not have any modules: the cyclic relationships become too complex to model simply, and it is unlikely that any such module can exist. For cases where the desired dependence is not listed, we would suggest that approximating it with a Taylor polynomial of some finite degree could suffice.

This module replicates the input quantity so that it can be used as input to multiple (n) subsequent modules.

$$x \rightarrow y_1 + y_2 + \ldots + y_n. \tag{12}$$

Linear:

$$\boxed{\alpha|y|_\infty = \beta|x|_0}$$

This module produces a quantity of an output type that is proportional to the quantity of an input type. For integer coefficients α and β, the module contains a single reaction:

$$\alpha x \rightarrow \beta y. \tag{13}$$

Multiplication:

$$\boxed{|y|_\infty = |x|_0 \times |z|_0}$$

This module will take two quantities and multiply them together. Initially there should be no y or a molecules present. The following bit of pseudo-code explains the method used to multiply the two quantities:

```
ForEach z:
   Y += X;
```

The reactions are:

$$z \xrightarrow{\text{slow}} a \tag{14}$$

$$a + x \xrightarrow{\text{faster}} a + x' + y \tag{15}$$

$$a \xrightarrow{\text{fast}} \varnothing \tag{16}$$

$$x' \xrightarrow{\text{medium}} x. \tag{17}$$

This module will destroy all of type z, but conserve the quantity x. Note also, that z need not be present in full quantity at the beginning of calculation, while x must.

Exponentiation:

$$\boxed{|y|_\infty = 2^{|x|_0}}$$

This module consumes molecules of an input type one at a time, doubling the quantity of an output type for each. Its behavior is described by the following pseudo code:

```
ForEach x:
  Y = 2 * Y;
```

The reactions are:

$$x \xrightarrow{\text{slow}} a \tag{18}$$

$$a + y \xrightarrow{\text{faster}} a + 2y' \tag{19}$$

$$a \xrightarrow{\text{fast}} \varnothing \tag{20}$$

$$y' \xrightarrow{\text{medium}} y. \tag{21}$$

Initially, $|y|$ must be one and all other quantities (except $|x|$) are zero.

Figure 1 shows how internal rate separation plays a large role in keeping the system working smoothly. If the input is allowed to achieve very large values, the amount of rate separation required can become quite large.

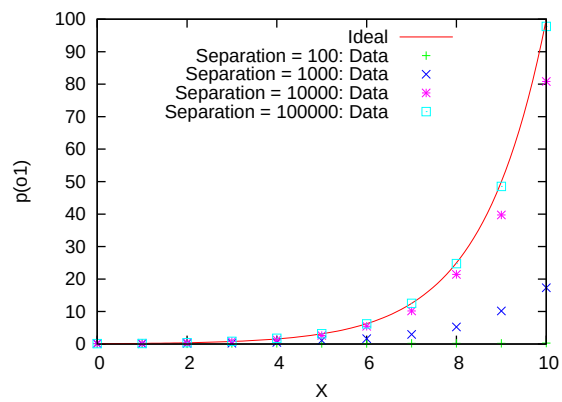

Figure 1: The Exponentiation module - Monte Carlo simulations were performed on an exponentiation module feeding an incorporation module that fed a stochastic module, the result of which was a sweep of the output from 0 to 100%. The exact mathematical form of this equation is shown for comparison.

Logarithm:

$$\boxed{|y|_\infty = log_2(|x|_0)}$$

This module is similar to the exponentiation module, except that instead of doubling the output, the input is forced to halve itself; each time it does so, the output is incremented by one. Its behavior is described by the following pseudo-code:

```
While Not(X==1):
  X = ceiling(X/2);
  Y = Y+1;
```

The reactions are:

$$b \xrightarrow{\text{slow}} a + b \tag{22}$$

$$a + 2x \xrightarrow{\text{faster}} c + x' + a \tag{23}$$

$$2c \xrightarrow{\text{faster}} c \tag{24}$$

$$a \xrightarrow{\text{fast}} \varnothing \tag{25}$$

$$x' \xrightarrow{\text{medium}} x \tag{26}$$

$$c \xrightarrow{\text{medium}} y. \tag{27}$$

Initially, $|b|$ is a small but non-zero quantity and all other quantities (except $|x|$) are zero. Figure 2 shows how this module compares to the function it attempts to emulate. Due to the integer precision of each "halving" (and the fact that the algorithm always rounds up) the result is actually the ceiling function of the desired logarithm.

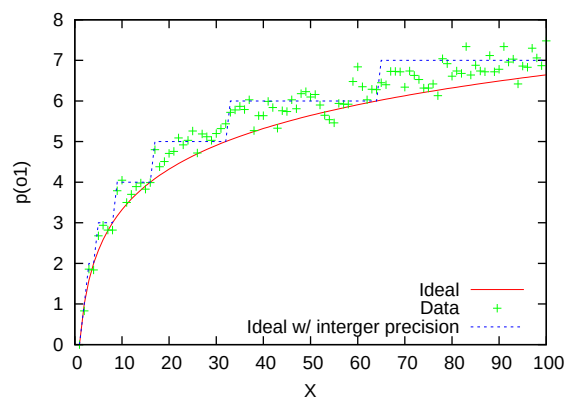

Figure 2: The Logarithm module - Monte Carlo simulations were performed on a logarithm module feeding an incorporation module. The exact mathematical form of this equation is shown for comparison.

Raising to a Power:

$$\boxed{|y|_\infty = |x|_0^{|p|_0}}$$

This module implements the raising of an input to a power based on the computations $X^P = \prod_P(X)$. So we must loop over multiplying X by a running product (α): $\alpha X = \sum_X(\alpha)$. Since multiplication is already a loop, this implies a double loop:

```
ForEach p:
   ForEach x:
      D = D + Y;
   Y = D;
   D = 0;
```

The reactions are:

$$p \xrightarrow{\text{slowest}} a \tag{28}$$
$$a + x \xrightarrow{\text{medium}} b + a + x' \tag{29}$$
$$b + y \xrightarrow{\text{fastest}} y' + d + b \tag{30}$$
$$b \xrightarrow{\text{faster}} \varnothing \tag{31}$$
$$y' \xrightarrow{\text{fast}} y \tag{32}$$
$$a \xrightarrow{\text{slow}} e \tag{33}$$
$$e + x' \xrightarrow{\text{faster}} e + x \tag{34}$$
$$e + y \xrightarrow{\text{faster}} e \tag{35}$$
$$e \xrightarrow{\text{fast}} \varnothing \tag{36}$$
$$d \xrightarrow{\text{slower}} y. \tag{37}$$

Initially, Y is one and all other quantities (excluding X and P) are zero. Reaction 28 starts the outer loop, Reaction 29 through Reaction 32 are simply the multiplication module with the first reaction gated by the outer loop molecule (a). Reaction 33 removes the loop molecule (a) and replaces it with a reset molecule (e). Reaction 34 resets $|x|$ for the next iteration of the loop. Reaction 35 sets $|y|$ to zero, so Reaction 37 can set it to the current running product after Reaction 36 has removed the reset molecule (e).

Isolation:

$$\boxed{|y|_\infty = 1}$$

This module is used to enforce an initial state consisting of a single molecule of some type. It is needed as a precursor for exponentiation and raising to a power. The reactions are:

$$c + 2y \xrightarrow{\text{fast}} c + y \tag{38}$$

$$c \xrightarrow{\text{slow}} \varnothing. \tag{39}$$

The module requires only that the quantities of types y and c be non-zero at the outset. Upon completion, there is exactly one molecule of type y and none of type c. Note that the molecules of c are all consumed, so the molecules of y can serve as inputs to other modules, provided that Reaction 39 completes in time.

2.1.2 Combining Modules

Note that in our definitions above, the molecular types are specific to each module (e.g., each x appearing in a different module should be considered a distinct type when combining these). Also, the rates — "fast" vs. "slow" — are relative *within* the modules. When combining modules, one might have to choose reactions with appropriate separations in their rates. (In some cases, the slowest reaction in one module might be *faster* than the fastest reaction in the next.)

Also note that with the linear, fan-out, subtraction, and multiplication modules, our scheme can be used to implement arbitrary polynomial functions; thus, in principle, it could be used to approximate complex functions through Taylor series expansions.

Example 2 Cubic Polynomial

Suppose that one desires output quantities that are equal to the quantity, its square, and its cube, of an input quantity ($|y_1| = |x|, |y_2| = |x|^2, |y_3| = |x|^3$). The simplest way to accomplish this would be to string together a pair of multiplication modules, and feed them

with the output of a fan-out module:

$$x \xrightarrow{\text{fastest}} x_1 + x_2 + y_1 \tag{40}$$
$$x_1 \xrightarrow{\text{medium}} a \tag{41}$$
$$a + y_1 \xrightarrow{\text{fastest}} a + y_1' + y_2 \tag{42}$$
$$a \xrightarrow{\text{faster}} \varnothing \tag{43}$$
$$y_1' \xrightarrow{\text{fast}} y_1 \tag{44}$$
$$x_2 \xrightarrow{\text{slower}} b \tag{45}$$
$$b + y_2 \xrightarrow{\text{faster}} b + y_2' + y_3 \tag{46}$$
$$b \xrightarrow{\text{fast}} \varnothing \tag{47}$$
$$y_2' \xrightarrow{\text{medium}} y_2. \tag{48}$$

The first reaction quickly produces quantities x_1, x_2 and the output y_1 all equal to the input quantity. The next four reactions multiply two of these, x_1 and y_1 in such a way that y_1 is not consumed. The result of this is the quantity of y_2 produced is equal to the square of the input quantity. The final four reactions again multiply this squared quantity with the input, producing a quantity of y_3 that is the cube of the input quantity.

From here it is a trivial matter to create any integer cubic polynomial by applying linear scaling to each of these quantities to create a final output. By having the output of these modules be of the same type, we achieve addition, if we need subtraction the outputs can be a common 'negative' type(y_n) which will then be combined with the output type in a subtraction module:

$$\alpha_1 y_1 \xrightarrow{\text{slowest}} \beta_1 y \tag{49}$$
$$\alpha_2 y_2 \xrightarrow{\text{slowest}} \beta_2 y_n \tag{50}$$
$$\alpha_3 y_3 \xrightarrow{\text{slowest}} \beta_3 y \tag{51}$$
$$y + y_n \xrightarrow{\text{fast}} \varnothing. \tag{52}$$

□

2.2 Module Locking

We discuss schemes for locking the looping mechanisms in modules as well as locking successive modules that are chained together[2].

2.2.1 Inter-Module Locking

To implement more complex biochemical computation, the simple modules outlined in Section 2.1 can be nested, with one module performing an arithmetic operation on an input and passing it to the next module. This generally requires the first module to complete execution prior to the second starting. This can be accomplished by creating modules with varying speeds as in Example 2. However, because a "faster" module needs to be a few orders of magnitude faster than the "slower" one, it only requires chaining a few together before we end up with rates that vary from nanoseconds to days, which is not ideal. The alternative is for the first module to prevent subsequent modules from firing until it is finished, we call this process "locking."

The first thing to do when locking a module is to modify the primary reaction of the module (the reaction that must occur first) to require an extra reactant. This locks the module until the new reactant, which we call a "key" is present. Next, we must identify a set of "indicator" molecules, those molecules whose presence indicates that the module being locked should remain so, *i.e.* the key should not be produced in the presence of an indicator. Then, we add the following reactions:

$$\varnothing \xrightarrow{\text{slow}} \text{keysmith}, \quad \text{keysmith} \xrightarrow{\text{slow}} \text{key}. \tag{53}$$

The keysmith molecule is simply a new chemical type that allows for the creation of a key. This creates a two step process to allow the key for a module to be created. Finally, to prevent the key being created in the presence of an indicator, we add the reaction

$$\text{indicator} + \text{keysmith} \xrightarrow{\text{fast}} \text{indicator} \tag{54}$$

for each indicator molecule. This prevents the creation of a key by destroying the keysmith before it can create a key when any indicator molecule is present. We illustrate this with an example.

Example 3 Locked Linear Module

Consider the simple case of a "linear" module,

$$|y| = |x|.$$

Suppose that the output of this module is the input to a stochastic module (as discussed in) that produces an outcome A with probability $p_1 = y/100$ and an outcome B with

probability $p_2 = (100 - y)/100$. Without locking, the reactions are:

$$\begin{aligned}
x &\xrightarrow{\text{fast}} y \\
y + e_2 &\xrightarrow{\text{fast}} e_1 \\
e_1 &\xrightarrow{\text{slow}} d_1 \\
e_2 &\xrightarrow{\text{slow}} d_2 \\
e_1 + d_1 &\xrightarrow{\text{fast}} 2d_1 \\
e_2 + d_2 &\xrightarrow{\text{fast}} 2d_2 \\
d_1 + d_2 &\xrightarrow{\text{fastest}} \varnothing \\
d_1 + &\xrightarrow{\text{slow}} d_1 + A \\
d_2 + &\xrightarrow{\text{slow}} d_2 + B.
\end{aligned} \tag{55}$$

Here e_1 and e_2 are initialized to 0 and 100, respectively. Modifying the reactions such that the stochastic module is locked until the linear module completes, the reactions are:

$$\begin{aligned}
x &\xrightarrow{\text{slow}} y \\
y + e_2 &\xrightarrow{\text{slow}} e_1 \\
\text{key} + e_1 &\xrightarrow{\text{slow}} \text{key} + d_1 \\
\text{key} + e_2 &\xrightarrow{\text{slow}} \text{key} + d_2 \\
e_1 + d_1 &\xrightarrow{\text{fast}} 2d_1 \\
e_2 + d_2 &\xrightarrow{\text{fast}} 2d_2 \\
d_1 + d_2 &\xrightarrow{\text{fastest}} \varnothing \\
d_1 + &\xrightarrow{\text{slow}} d_1 + A \\
d_2 + &\xrightarrow{\text{slow}} d_2 + B \\
\varnothing &\xrightarrow{\text{slow}} \text{keysmith} \\
\text{keysmith} &\xrightarrow{\text{slow}} \text{key} \\
x + \text{keysmith} &\xrightarrow{\text{fast}} x \\
y + \text{keysmith} &\xrightarrow{\text{fast}} y.
\end{aligned} \tag{56}$$

Here we have added four reactions, two for the key generation and one for each of the indicator molecules, x and y.

Figure 3 compares the locked version to the original version. Curves of the probabilistic response for outcome A are plotted for different rates. We see how effective locking is, even if "fast" is only twice as fast as "slow." □

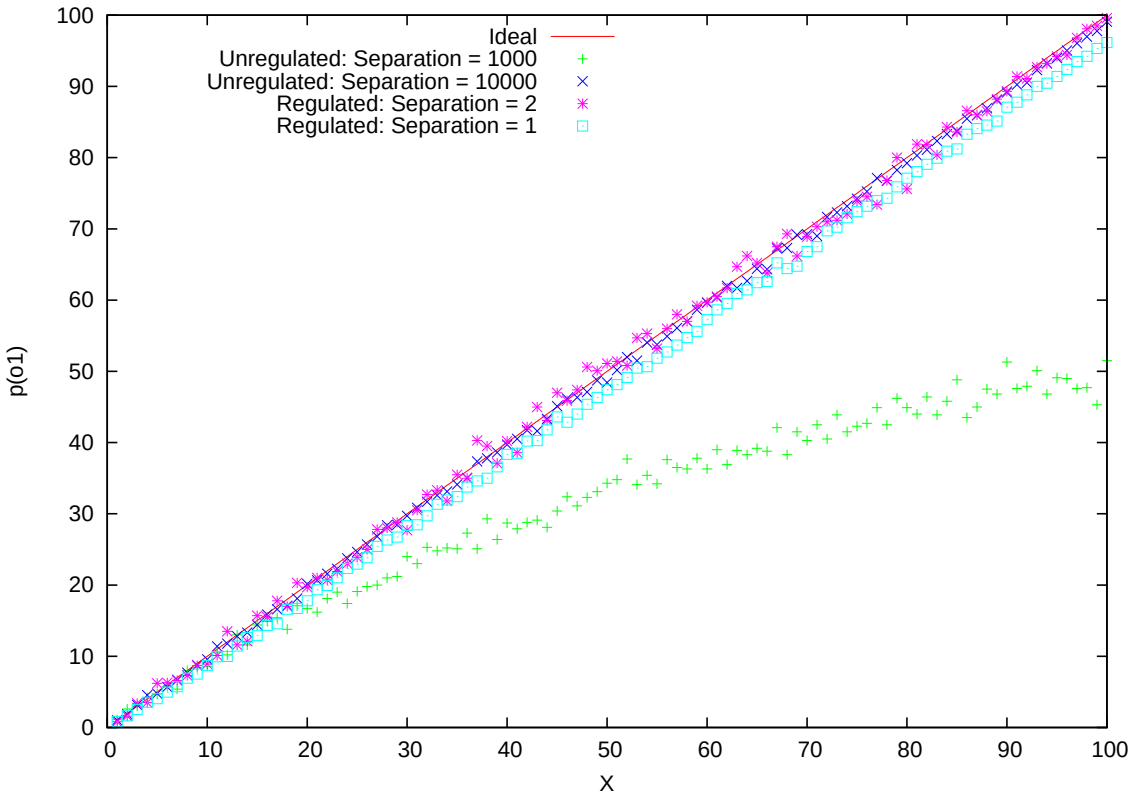

Figure 3: Inter-module Locking – the probabilistic response of the locked and unlocked versions of the modules in Example 3. The first is for the unlocked version with the rates "slow," "fast," and "fastest," each separated by a factor of 1000; the next with these rates separated by a factor of 10,000; the next for the locked version with rates "slow" and "fast" separated by a factor of 2; and the last with these rates identical.

Similar examples are shown for both logarithm and exponentiation in Figure 4 and Figure 5, respectively. While the low quantities involved in the logarithm example do not play to the strength of this scheme, the exponentiation example shows how even with *no* rate separation, a better approximation to the true value is reached than with a rate separation of 1000 in the unlocked case.

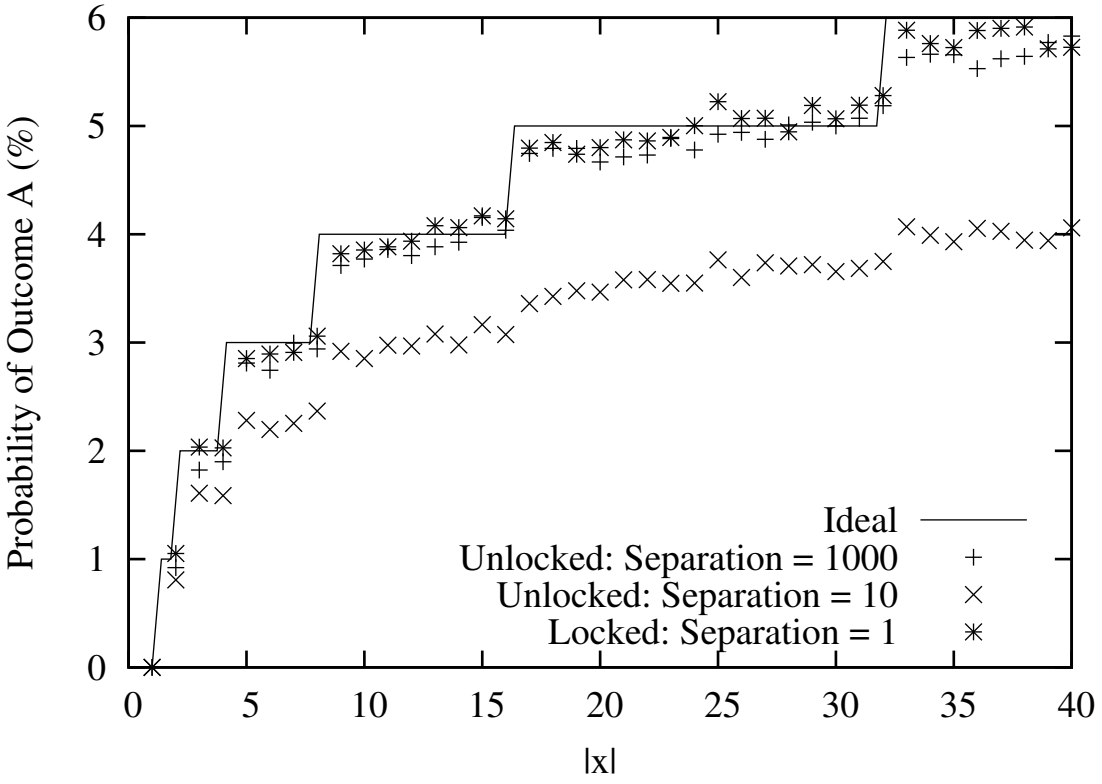

Figure 4: Inter-module Locking (Logarithm) – an example with a logarithm module feeding a stochastic module; the correct value of the probability is the ceiling of the base two logarithm in percent (shown as ideal).

2.2.2 Intra-Module Locking

Most of the functional modules in Section 2.1 (those with more than one or two reactions) operate with a looping construct that iteratively works toward the correct answer. In all such modules, it is important that the reactions fire in the correct order. In addition to unlocking the reactions when it is time for them to execute, we must also have the ability to "re-lock" them when it is time for them to stop executing.

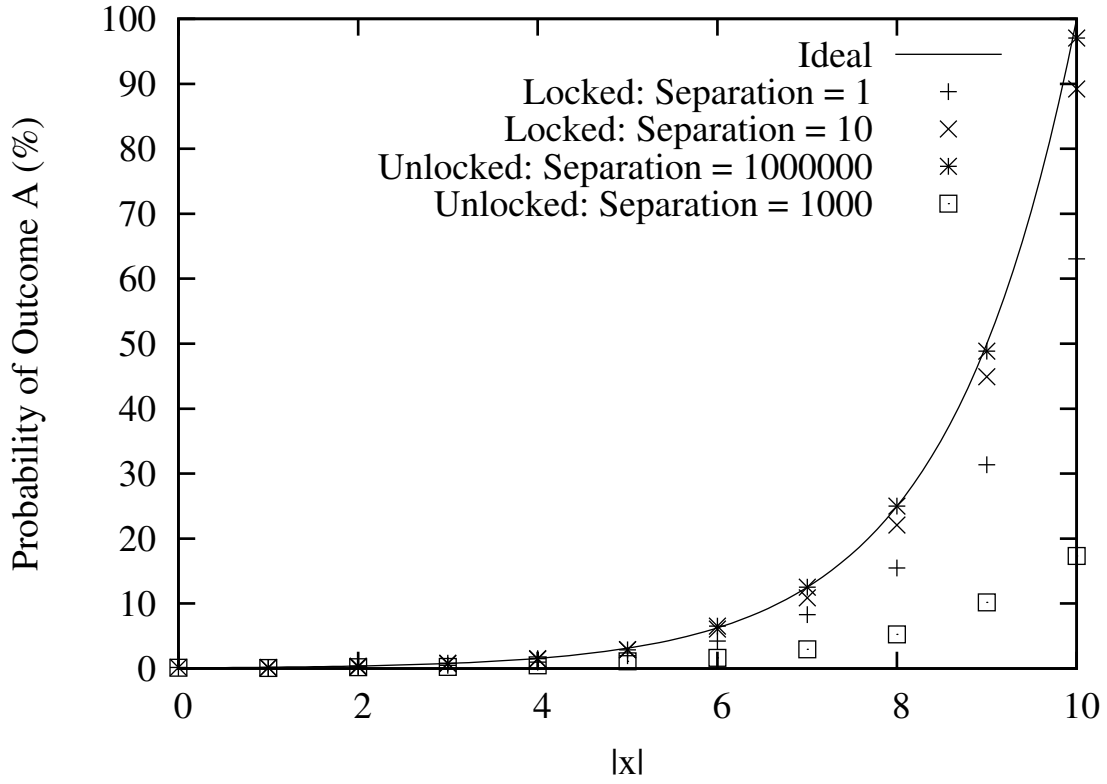

Figure 5: Inter-module Locking (Exponentiation) – an example with an exponentiation module feeding a stochastic module; the correct value of the probability is proportional to the exponential (base 2) of the quantity of the input, hitting 100% at an input of 10.

Our scheme involves adding a key requirement to most of the reactions in the loops of modules. 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 set of reactions for this functionality is:

$$\begin{aligned} \forall i: \quad & \varnothing \xrightarrow{\text{slow}} \text{keysmith}_i \\ \forall i: \quad & \text{keysmith}_i \xrightarrow{\text{slow}} \text{key}_i \\ \forall i: \quad & \text{key}_i \xrightarrow{\text{slow}} \varnothing \\ \forall i,j: \quad & \text{key}_i + \text{keysmith}_j \xrightarrow{\text{fast}} \text{key}_i. \end{aligned} \tag{57}$$

Example 4 Locked Multiplication Module

Reaction set 58 gives a locked version of the multiplication module from Section 1.3. The lines separate the reactions into three sets: the original reactions, the generic module-locking reactions, and some locking reactions specific to multiplication.

$$
\begin{array}{rcl}
\text{key}_1 + x & \xrightarrow{\text{slow}} & a \\
a + y & \xrightarrow{\text{slow}} & a + y' + z \\
\text{key}_2 + a & \xrightarrow{\text{slow}} & \varnothing \\
\text{key}_3 + y' & \xrightarrow{\text{slow}} & \text{key}_3 + y \\
\hline
\varnothing & \xrightarrow{\text{slow}} & \text{keysmith}_1 \\
\varnothing & \xrightarrow{\text{slow}} & \text{keysmith}_2 \\
\varnothing & \xrightarrow{\text{slow}} & \text{keysmith}_3 \\
\text{keysmith}_1 & \xrightarrow{\text{slow}} & \text{key}_1 \\
\text{keysmith}_2 & \xrightarrow{\text{slow}} & \text{key}_2 \\
\text{keysmith}_3 & \xrightarrow{\text{slow}} & \text{key}_3 \\
\text{key}_1 & \xrightarrow{\text{slow}} & \varnothing \\
\text{key}_2 & \xrightarrow{\text{slow}} & \varnothing \\
\text{key}_3 & \xrightarrow{\text{slow}} & \varnothing \\
\text{key}_1 + \text{keysmith}_1 & \xrightarrow{\text{fast}} & \text{key}_1 \\
\text{key}_1 + \text{keysmith}_2 & \xrightarrow{\text{fast}} & \text{key}_1 \\
\text{key}_1 + \text{keysmith}_3 & \xrightarrow{\text{fast}} & \text{key}_1 \\
\text{key}_2 + \text{keysmith}_1 & \xrightarrow{\text{fast}} & \text{key}_2 \\
\text{key}_2 + \text{keysmith}_2 & \xrightarrow{\text{fast}} & \text{key}_2 \\
\text{key}_2 + \text{keysmith}_3 & \xrightarrow{\text{fast}} & \text{key}_2 \\
\text{key}_3 + \text{keysmith}_1 & \xrightarrow{\text{fast}} & \text{key}_3 \\
\text{key}_3 + \text{keysmith}_2 & \xrightarrow{\text{fast}} & \text{key}_3 \\
\text{key}_3 + \text{keysmith}_3 & \xrightarrow{\text{fast}} & \text{key}_3 \\
\hline
\text{keysmith}_1 + y' & \xrightarrow{\text{fast}} & y' \\
\text{keysmith}_1 + a & \xrightarrow{\text{fast}} & a \\
\text{keysmith}_2 + y & \xrightarrow{\text{fast}} & y \\
\text{keysmith}_3 + a & \xrightarrow{\text{fast}} & a.
\end{array}
\tag{58}
$$

1. In the first set, notice that we do not add a lock to the second reaction; this is because the reaction is already locked by the "looping" type a. Notice, also, that the first and third reactions destroy the keys that they require. This prevents them from firing more than once. The fourth reaction does not destroy its key, since it fires repeatedly.

2. In the second set, the first three reactions produce the appropriate keysmiths; the

next three allow those keysmiths to create keys; the next three cause keys that are no longer needed to disappear; the next nine ensure that there are no keysmiths left to create keys when there is already one in the system.

3. The third set of reactions ensure that the wrong key will not be produced. The requirement is that the keysmith of a locked reaction should not be allowed to be created when any of the other (related) locked reactions could be firing. We accomplish this by destroying the keysmith associated with any key that should not be created.

□

In general, most modules employing looping constructs will have four parts to the loop: a loop initiator, loop actions, a loop closer, and a loop reset. The loop actions are all reactions that require, but do not destroy, the looping type (a in the example above); no changes need to be made to lock these reactions. The loop initiator creates the looping type, while the loop closer destroys it. Both of these parts require a key that is destroyed by the reactions because they should only occur once per loop. The loop-reset reactions do not involve the loop type in any way; each will require a third, shared key, but they need not destroy it.

Both generic module locking reactions (i.e., those in Module 57) as well as some reactions specific to the module must be added. These include reactions for:

- destroying the keysmith for the loop initiator if the looping molecule is present,
- destroying the keysmith for the loop initiator if any loop reset reactions can fire,
- destroying the keysmith for the loop closer if any loop actions can take place, and
- destroying the keysmith for the loop reset reactions if the looping molecule is present.

With these modifications, the only requirement on the rate of the reactions is that all reactions that destroy a keysmith be "fast" and the others "slow."

Tables 1, 2, and 3 compare the accuracy of the locked vs. unlocked versions of various functional modules using different separations in the rate constants. For the unlocked version, we used the rates $1, \lambda, \lambda^2$, and λ^3 as the values for "slowest," "slow," "fast," and "fastest." For the locked modules, we used 1 and λ for "slow" and "fast." Note that the total range of rates in the unlocked case is λ^3. For a fair comparison, we defined the "accuracy gain" for the scheme to be the error of the unlocked method at $\lambda = 10$ divided by

Table 1: A Comparison of the Accuracy of the Locked and Unlocked Versions of Multiplication Module.

Calculation: 10×10		
	%error	
λ	unlocked	locked
1	77.75%	48.46%
10	27.07%	24.67%
100	4.20%	3.09%
1000	0.45%	0.33%
10000	0.05%	0.02%
# of reactions	4	26
Accuracy gain: 82.03×		

Table 2: A Comparison of the Accuracy of the Locked and Unlocked Versions of Exponentiation Module.

Calculation: 2^5		
	%error	
λ	unlocked	locked
1	75.02%	N/A
10	18.027%	N/A
100	2.37%	33.72%
1000	0.25%	2.58%
10000	0.02%	0.26%
# of reactions	4	26
Accuracy gain: 6.99×		

Table 3: A Comparison of the Accuracy of the Locked and Unlocked Versions of Logarithm Module.

Calculation: $log_2(64)$		
	%error	
λ	unlocked	locked
1	267.03%	169.99%
10	41.36%	69.89%
100	5.24%	11.57%
1000	0.53%	1.27%
10000	0.05%	0.14%
# of reactions	6	30
Accuracy gain: 32.56×		

the error of the locked scheme at $\lambda = 1000$, since both sets of reactions would then require that the fastest reaction be 1000 times faster than the slowest.

It is interesting to note that the unlocked versions of multiplication and exponentiation tend to under-compute the result, whereas our locked versions tend to over-compute it. This is because, in the unlocked case, the error that occurs is removing the loop molecule prematurely; in the locked case, the error comes from allowing the loop to reset while active.

2.3 Stochastic Module: Quantities to Probabilities

The stochastic module allows for a choice between multiple outcomes. This choice is picked randomly according to probabilities that are set by the user. It takes, as inputs, quantities of various chemical species; from these quantities, it derives probabilities for progressing to any one of the given ending states. It then will pick one of these ending states based on the probabilities that it finds. Once an ending state is picked, the system will advance toward this ending state.

The method that it uses is based on one simple premise: calculating the probabilities for a single reaction is easy, while calculating the probabilities of ending at some state after even a small number of reactions is hard. Using this knowledge, we will confine the choice to a single reaction event. We confine the number of possible reactions to a small number (equal to the number of ending states), and we will set the input quantities to force the probabilities of picking any given reaction out of that set equal to the desired probability of ending in a given state. The final step is to ensure (to within a small chance of error) that this single reaction event dictates the course of the system and pushes it toward the desired outcome.

Example 5 Stochastic Module with Three Outcomes

Suppose that we have a system with molecular types d_1, d_2, and d_3. We wish to program the production of these types with the probability distribution

$$p_1 = 0.3, \qquad p_2 = 0.4, \qquad p_3 = 0.3,$$

respectively. To do so, we set up *initializing* reactions:

$$e_1 \xrightarrow{1} d_1, \quad e_2 \xrightarrow{1} d_2, \quad e_3 \xrightarrow{1} d_3.$$

We initialize the system with quantities of e_1, e_2, and e_3 in the desired ratio of $3:4:3$,

$$|e_1| = 30, \qquad |e_2| = 40, \qquad |e_3| = 30.$$

Should we want a different probability distribution, we simply change the ratio of these initial quantities. Given these ratios, the reaction producing d_1 fires first with probability 0.3, the one producing d_2 fires first with probability 0.4, and the one producing d_3 fires first with probability 0.3. Since these are the same probabilities we desire at the conclusion, we want to cement this initial choice. Accordingly, we set up *reinforcing* reactions:

$$e_1 + d_1 \xrightarrow{10^3} 2d_1, \quad e_2 + d_2 \xrightarrow{10^3} 2d_2, \quad e_3 + d_3 \xrightarrow{10^3} 2d_3.$$

In addition, we set up *stabilizing* reactions:

$$\begin{aligned} d_1 + e_2 &\xrightarrow{10^3} d_1, \quad d_1 + e_3 \xrightarrow{10^3} d_1, \quad d_2 + e_1 \xrightarrow{10^3} d_2, \\ d_2 + e_3 &\xrightarrow{10^3} d_2, \quad d_3 + e_1 \xrightarrow{10^3} d_3, \quad d_3 + e_2 \xrightarrow{10^3} d_3. \end{aligned}$$

Note that the reinforcing and stabilizing reactions have much higher rates than the initializing reactions, thus they will fire more rapidly (if able) than a reaction having similar reactant types. Finally, we set up *purifying* reactions:

$$d_1 + d_2 \xrightarrow{10^6} \varnothing, \quad d_1 + d_3 \xrightarrow{10^6} \varnothing, \quad d_2 + d_3 \xrightarrow{10^6} \varnothing.$$

Note that the purifying reactions have yet higher rates.

The reinforcing, stabilizing, and purifying reactions ensure that as soon as an initializing reaction fires, producing a molecule of d_i, this choice quickly wins out: the production of more molecules of d_i is encouraged, while the production of the other types d_j, $j \neq i$, is strongly inhibited. As such, the firing probabilities for the initializing reactions at the outset dictate the probability distribution of the final outcome. □

2.3.1 The Set of Reactions

The stochastic module consists of five categories of reactions: Initializing, Reinforcing, Stabilizing, Purifying, and Working. The rates are assumed to be similar for all the reactions in each category; however, the rates between categories must be different: some categories are slow and other are comparatively fast, as is explained in Section 2.3.3.

Since controlling the probability of an arbitrary path through the system is difficult, we

aim to require control over only a single reaction. The remaining reactions work to cement the decision made in this one reaction and to perform the function dictated by it.

For all of the categories, the subscripts i and j run over the number of desired outcomes. For each outcome, we have an *input* type e, a *catalyst* type d, *output* types o, and *food* types f which limit the output, once one is decided.

Initializing Reactions

$$\forall\, i: \qquad e_i \xrightarrow{k_i} d_i \tag{59}$$

These reactions initiate the response with the production of a catalyst type. They are the slowest reactions in the system. The first one to fire generally determines the outcome. (As is discussed in Section 2.3.3, the likelihood of a different outcome is vanishingly small.)

Reinforcing Reactions

$$\forall\, i: \qquad d_i + e_i \xrightarrow{k'_i} 2d_i \tag{60}$$

These reactions amplify the choice made by the initializing reactions, increasing the quantity of the catalyst type. (The quantity of catalyst that is produced here is limited by amount of the input type that is supplied. It could be limited some other way, but this is convenient.)

Stabilizing Reactions

$$\forall\, j \neq i: \qquad d_i + e_j \xrightarrow{k''_{ij}} d_i \tag{61}$$

These reactions consume all input types other than the one that was selected. So they inhibit competing outcomes.

Purifying Reactions

$$\forall\, j \neq i: \qquad d_i + d_j \xrightarrow{k'''_{ij}} \varnothing \tag{62}$$

These reactions quickly suppress any competing catalyst types. They are the fastest reactions in the system. If ever there are multiple catalyst types present, those in the minority

are quickly eliminated, whereas those in the majority are only slightly weakened in number.

Working Reactions

$$\forall i, \ell_i : \qquad d_i + f_{\ell_i} \overset{k_i''''}{\rightarrow} d_i + o_{\ell_i} \tag{63}$$

These reactions take the decision made by the initializing reactions and turn it into action: they produce output types in the desired quantity. Several output types in differing proportions can be created for each catalyst type. This can be accomplished with different working reactions operating on the same catalyst type. Alternatively, a single working reaction can be set up with multiple output types in the desired proportions.

2.3.2 Initial Quantities

The probability of the i-th initializing reaction firing first is proportional to its rate, k_i, and to the quantity ($|e_i|$) of its input type e_i. Accordingly, we can *program* the firing probabilities by setting the ratio of the initial quantities:

$$\forall\, i : \qquad p_i = \frac{k_i|e_i|}{\sum_{\forall j} k_j|e_j|}. \tag{64}$$

Thus, the initial quantities of the input types directly determine the probability distribution of the outcomes. At the outset, there are no catalyst types or output types. The initial quantities of the food types are set to the maximum quantity desired for the corresponding output types.

2.3.3 Reaction Rates

The rates should be selected so that the initializing and working reactions are the slowest, the reinforcing and stabilizing reactions comparatively much faster, and the purifying reactions fastest of all:

$$k_i \approx k_i'''' \quad \ll \quad k_i' \approx k_{ij}'' \quad \ll \quad k_{ij}'''.$$

In order to quantify the effect of this separation in the rates, let us choose a multiplicative factor, γ, and set the rates as follows:

$$\gamma k_i = k_i' = k_{ij}'' = k_{ij}'''/\gamma = \gamma k_i''''. \tag{65}$$

If we define an *error* to be the case where the first initializing reaction to fire does *not* determine the final outcome; instead, a different catalyst type wins out. We characterize this error as a function of γ. More specifically, we set up the reactions described in Section 2.3.1 for $i = 1, 2, 3$, with each $k_i = 1$ and each $k'_i, k''_{ij}, k'''_{ij}$, and k''''_i set according to Equation 65. We set the initial quantity of each input type to 100. We assume that a working reaction needs to fire 10 times for us to declare an outcome. We performed Monte Carlo simulations and obtained the results shown in Figure 6. The graph shows that the error can be made vanishingly small by increasing the separation in the rates.

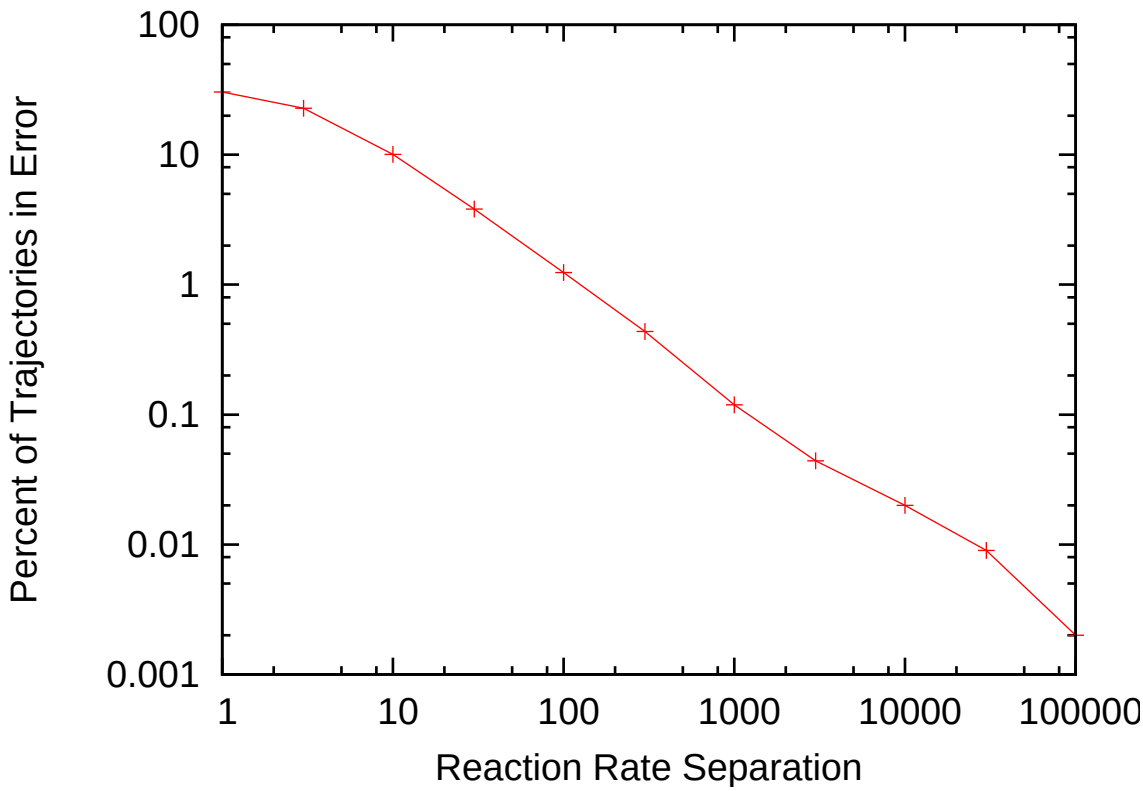

Figure 6: Error Analysis for the Stochastic Module. Monte Carlo simulations with 100,000 trials were performed for different values of γ. The graph gives the percentage of trials that resulted in error.

2.3.4 Locking the Stochastic module

The above method, while workable, may require rather high rate separations to accomplish a specified error rate. Here we propose an alternative approach, based on locking. All

initialization reactions share one key; as such, the choice between the reactions is made independently of the types and quantities of keys present. Thus, the probability distribution is still a function of the input types.

Our stochastic module becomes:

$$\begin{aligned}
& & \varnothing &\xrightarrow{\text{slow}} \text{keysmith} \\
& & \text{keysmith} &\xrightarrow{\text{slow}} \text{key} \\
\forall i: & & \text{key} + e_i &\xrightarrow{\text{slow}} d_i \\
\forall i: & & d_i + e_i &\xrightarrow{\text{fast}} 2d_i \\
\forall i \neq j: & & d_i + e_j &\xrightarrow{\text{fast}} d_i \\
\forall i: & & d_i + \text{keysmith} &\xrightarrow{\text{fast}} d_i \\
\forall i: & & d_i &\xrightarrow{\text{slow}} d_i + O_i.
\end{aligned} \tag{66}$$

The lock on each initiating reaction ensures that only a single random choice can be made. Once made, this choice inhibits all other choices both by consuming competing molecules and by destroying subsequent keysmiths. It is interesting to note that this version actually requires *fewer* reactions than our previous version for cases with five or more outcomes.

Figure 7 shows that the error at any given rate separation is more than an order of magnitude *lower* with locking than without; this is before taking into account that the unlocked version actually requires three levels, thus needing two such separations, while the locked version needs only one. Both the locked and unlocked versions were of a stochastic module with three outcomes; 100,000 random trajectories were run for each data point. With fewer requirements on reaction rates, much less error for a given separation in the rates and fewer required reactions in cases with large numbers of outcomes, the locked version is clearly superior.

3 Automation

In this section, we discuss how using *linear and integer programming* (LP and IP) techniques, the process of picking reactions to fit a desired stochastic response, can be automated. Linear and integer programming allows us to find an "optimal" solution to a problem with infinite solutions. These techniques require a set of constraints on a set of variables and an expression that must be either minimized or maximized within the constraints.

Here, we assume that we are attempting to create a system that produces a stochastic

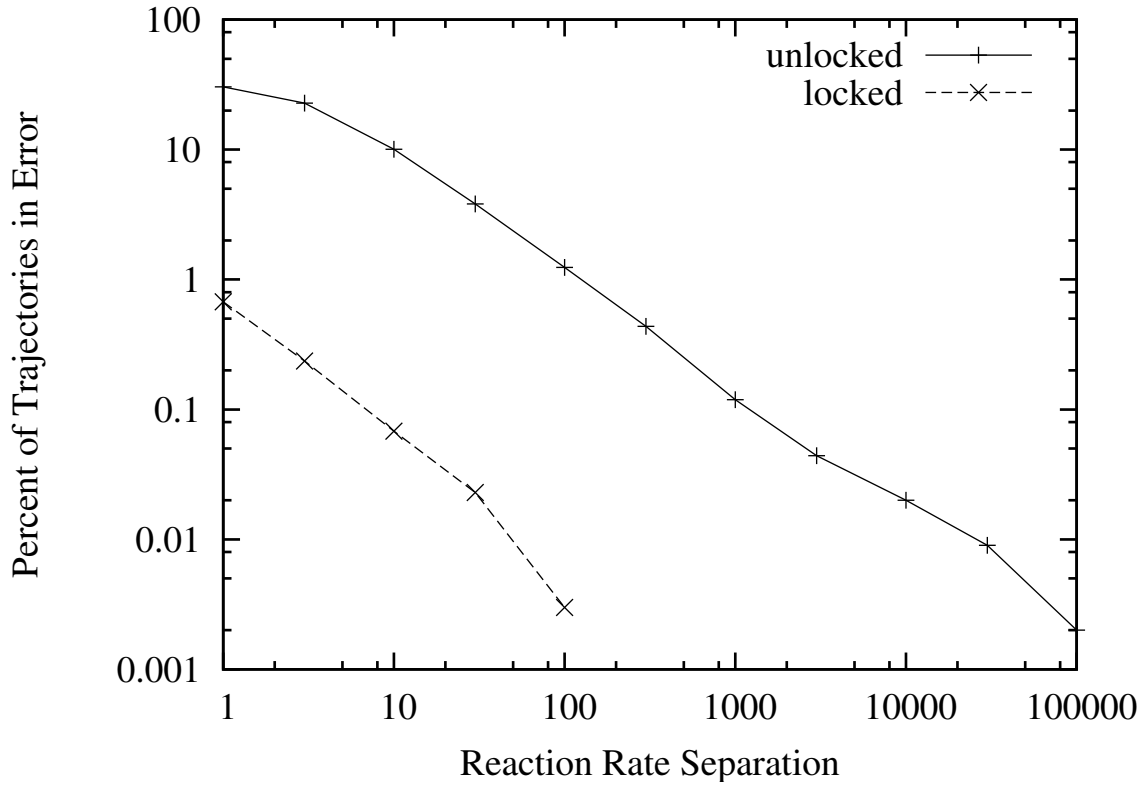

Figure 7: Accuracy of the Locked vs. Unlocked Stochastic Modules – The percent of trajectories whose initial choice is not reflected in the final state is shown as a function of the separation between "fast" and "slow" rates.

response that is some function of a set of input quantities ($\vec{X}$). Recall from Section 2.3 that the probability of picking each outcome is given by

$$\forall\, i: \qquad p_i = \frac{E_i(\vec{X})k_i}{\sum_{\forall j} E_j(\vec{X})k_j}, \tag{67}$$

where $E_n(\vec{X})$ is the value $|e_n|$ that is produced for an input set $\vec{X}$. To obtain any desired set of probabilities for any given input $\vec{X}$, we need to set the ratio of the various e types. Obviously then, any multiple of a solution is itself a solution for any given input. Additionally, this equation can be solved independently for any value of $\vec{X}$. We will use this fact to find a set of reactions that fit a desired probability distribution with respect to $\vec{X}$ in two parts: first, solving individual points for a single set of quantities $\vec{E}(\vec{X})$ that yields the desired output probabilities, and second, fitting those together with scaling to find the coefficients of a function that will fit them.

The set of points ($\vec{X}$s) to be solved and then fit are part of the specification. There is a trade-off to be made between calculation time and accuracy of fit, which both increase as the number of points increases.

3.1 Solving at a Single Point

For any given input $\vec{X}$, the desired output probabilities are known. Thus, we can solve for a set of quantities $\vec{E}$ that yields it. Since there is an infinite number of solutions (remember that all multiples of a solution are solutions), we will try to find the minimal solution. Since this will tend toward $\vec{0}$ (which gives us no information), we set a requirement that the sum of the quantities ($\sum_i |e_i|$) be greater than one.

Equation 67 can be rewritten as

$$\vec{p}(\vec{E}(\vec{X}) \cdot \vec{k}) = \vec{E}(\vec{X}) * \vec{k} \tag{68}$$

where we define the operations $\vec{a} \cdot \vec{b}$ and $\vec{a} * \vec{b}$ to be the dot product ($a_0b_0 + a_1b_1 \ldots$) and "element-wise" product ($< a_0b_0, a_1b_1 \ldots >$) of $\vec{a}$ and $\vec{b}$ respectively. Moving everything to one side we get

$$\vec{0} = \vec{p}(\vec{E}(\vec{X}) \cdot \vec{k}) - \vec{E}(\vec{X}) * \vec{k} = \vec{F}(\vec{X}). \tag{69}$$

Using this constraint directly can lead to rounding errors where a linear or integer programming solver will fail to find a solution. By allowing some error, we can avoid this

problem:

$$\Delta F_i \geq F_i(\vec{X}) \tag{70}$$

$$-\Delta F_i \leq -F_i(\vec{X}). \tag{71}$$

The error terms (ΔF_i) are to be minimized as part of our specification. This is the same as requiring the magnitude of $\vec{F}(\vec{X})$ to be small, which taken to the limit would be zero as required. Our constraints then become:

$$\Delta F_i \geq E_i k_i - p_i(\sum_j E_j k_j) \tag{72}$$

$$-\Delta F_i \leq -E_i k_i - p_i(\sum_j E_j k_j) \tag{73}$$

for each i (p_i and k_i are known).

We add the additional constraint

$$1 \leq \sum_i E_i, \tag{74}$$

to prevent a solution of $\vec{E} = \vec{0}$, and solve for $\vec{E}$ by minimizing

$$\sum_i E_i + w_{\Delta F} \sum_j \Delta F_j \tag{75}$$

where $w_{\Delta F}$ is a weighting factor that can be adjusted (ideally, it would be infinite, ensuring that all ΔFs where zero).

3.2 Fitting the Points

Once we have solutions ($\vec{E}(\vec{X})$) for many input points ($\vec{X}_1, ...\vec{X}_\ell$), we need to find a curve that fits them. The first step is to choose a function with which to fit the data; this function is allowed to be anything that can be created using the deterministic module (Section 2.1). For illustration, we show a cubic function of one variable quantity

$$\vec{G}(x) = \vec{c_0} + \vec{c_1}|x| + \vec{c_2}|x|^2 + \vec{c_3}|x|^3. \tag{76}$$

We then map this function to the solutions found for each $\vec{X}$, remembering that each point can be scaled independently

$$\vec{G}(\vec{X}_j) = m(\vec{X}_j)\vec{E}(\vec{X}_j). \tag{77}$$

As wtih Equation 69, Equation 77 is a constraint with an equality which can lead to rounding errors in practice, so we again try to minimize error:

$$\Delta G_i(\vec{X}_j) \geq G_i(\vec{X}_j) - m(\vec{X}_j)E_i(\vec{X}_j) \tag{78}$$

$$-\Delta G_i(\vec{X}_j) \leq m(\vec{X}_j)E_i(\vec{X}_j) - G_i(\vec{X}_j). \tag{79}$$

These constraints are for every term in $\vec{G}$ and for every solved point $\vec{X}_j$.

As before, the solution will collapse toward $\vec{0}$ so we require some non-zero term:

$$1 \leq \sum_j m(\vec{X}_j) \tag{80}$$

Finally, the term we must minimize is:

$$w_{\Delta G} \sum_{ij} \Delta G_i(\vec{X}_j) + w_m \sum_j m(\vec{X}_j) + \sum_{\vec{c} \in C} \|\vec{c}\|, \tag{81}$$

where C is the set of all coefficients in the function $\vec{G}$, and $w_{\Delta G}$ and w_m are weighting factors for deltas (again infinite is ideal) and the scaling factors respectively. With a properly tuned weighting factor w_m, a good fit can be made for the data points where Equation 67 becomes

$$\forall\, i: \qquad p_i = \frac{k_i G_i(\vec{X})}{\sum_j k_j G_j(\vec{X})}. \tag{82}$$

3.3 Integer Programming, Linear Programming and Time

Because we need to bring these numbers into the realm of reactions and quantities which each require integer quantities, we should be using integer programming for each of these steps, however, integer programming is NP-hard and thus slow. To speed things up we can substitute linear programming, but we loose the property that the solutions will be integers. This is perhaps not as bad as it sounds as we still obtain a solution, and we can scale solutions after the fact to create valid solutions with much more speed.

The following methods are listed in increasing calculation speed, and presumably decreased accuracy.

1. Use integer programming for the entire process.

2. Use linear programming to solve each point (the result will be scaled anyway) but in the second step require integer coefficients to the function $G(\vec{X})$.

3. Use linear programming for both steps adding a third step where the coefficients of $G(\vec{X})$ are scaled to integer quantities using the constraints

$$\Delta\bar{G}_i(\vec{X}_j) \geq \bar{G}_i(\vec{X}_j) - mG_i(\vec{X}_j) \tag{83}$$

$$-\Delta\bar{G}_i(\vec{X}_j) \leq mG_i(\vec{X}_j) - \bar{G}_i(\vec{X}_j) \tag{84}$$

   where $\bar{G}_i(\vec{X}_j)$ is a version of $G_i(\vec{X}_j)$ with integer coefficients. We then minimize the expression

$$w_{\Delta\bar{G}} \sum_{ij} \Delta\bar{G}_i(\vec{X}_j) + w_m m + \sum_{\bar{c}\in\bar{C}} \bar{c}, \tag{85}$$

   where $\bar{C}$ is the set of integer coefficients to $\bar{G}_i(\vec{X}_j)$, $w_{\Delta\bar{G}}$ is the weight for error and w_m is the weight for scaling.

4. Use linear programming for both steps, then scale the parameters by some constant and round to an integer.

3.4 Return to Reactions

Once the coefficients are found and converted into integers, we can convert them back into the reaction space. The first step is to create a stochastic module with the correct number of outcomes. The next step only requires knowledge of the function you used to fit the system and can be created before any calculation has been done. This step simply creates the quantities of your function that will be scaled by the unknown coefficients. The final step uses the information obtained through the integer programming technique to connect the two.

Example 6 Scaled Cubic polynomial

To implement Equation 76, we can simply chain two multiplication modules together:

$$\begin{aligned}
x &\xrightarrow{\text{faster}} y_1 + x_1 + x_2 \\
x_1 &\xrightarrow{\text{slow}} a_1 \\
a_1 + y_1 &\xrightarrow{\text{faster}} a_1 + y_1' + y_2 \\
a_1 &\xrightarrow{\text{fast}} \varnothing \\
y_1' &\xrightarrow{\text{medium}} y_1 \\
x_2 &\xrightarrow{\text{slower}} a_2 \\
a_2 + y_1 &\xrightarrow{\text{fast}} a_2 + y_2' + y_3 \\
a_2 &\xrightarrow{\text{medium}} \varnothing \\
y_2' &\xrightarrow{\text{slow}} y_2
\end{aligned}$$

which will produce y_1, y_2 and y_3 in quantities x, x^2 and x^3 respectively.

Next, the coefficients of the fitting equation are taken into account:

$$\begin{aligned}
y_1 &\xrightarrow{\text{slowest}} c_{11}e_1 + c_{21}e_2 + ... \\
y_2 &\xrightarrow{\text{slowest}} c_{12}e_1 + c_{22}e_2 + ... \\
y_3 &\xrightarrow{\text{slowest}} c_{13}e_1 + c_{23}e_2 + ...
\end{aligned}$$

This set of reactions is the only one that needs to be modified depending on the results of our computations. If a coefficient is negative, we instead produce that quantity of a "negative" type $\bar{e}_i$. We then set up reactions for the appropriate subtraction

$$e_i + \bar{e}_i \xrightarrow{\text{fast}} \varnothing \tag{86}$$

Finally, we set up a stochastic module that takes the 'e's as reactants to the initializing reactions and has the rate ratio assumed from the start.□

4 Theory to Practice

In this paper, we have discussed a method for creating sets of reactions that fit a desired behavior. However, real world dynamics do not allow us to simply create chemical species and dictate how they will interact. The interactions that exist between species is set by

the chemical properties of the species involved, thus we are limited to finding chemical species that will interact in the way we design. This is not a trivial exercise. Even if the interactions between all possible chemical types were known and tabulated (an impossible task), finding a subset within that master set of reactions that behaved as we designed and involved chemical species and that would not react in ways outside our definition would be computationally taxing, if it even existed.

It is our opinion that the only method to efficiently make this mapping from abstract types to real chemical species is to find a way to remap these reactions into DNA. By using predefined segments of DNA[15], it may be possible to create interactions that are, in fact, self contained and that perform according to the design. By finding ways to map the various modules of our scheme to actions, like inhibiting or triggering the transcription of a protein or set of proteins, that themselves may inhibit or trigger various other pathways, it may be possible to design real systems in a way that is practical and efficient.

5 Conclusion

Our methodology allows for a great amount of flexibility when creating a theoretical biochemical system. It allows for stochastic systems, deterministic systems and combinations of the two. The system produced is robust to the inherent randomness associated with such systems. The error produced by the random behavior is related to the rate separation between reactions of the various "levels" of speed. By employing locking mechanisms, it is possible to improve the error rate of systems with poor rate separation. This is done by including additional reactions, thus allowing for a trade off between error rate, reaction rate separation and reaction count. We also have shown how such systems could be designed in an automated fashion. We propose that, to bring this methodology out of the theoretical realm and into reality, the next step is to map these modules to DNA sequences.

References

[1] B. Fett, J. Bruck, M. Riedel, "Synthesizing Stochasticity in Biochemical Systems," *DAC*, pp. 640–645, 2007.

[2] B. Fett, M. Riedel, "Module locking in biochemical synthesis," *International Conference on Computer Aided Design*, pp. 758–764, 2008

[3] D. Endy and R. Brent, "Modelling Cellular Behaviour," *Nature*, Vol. 409, pp. 391–395, 2001.

[4] D. Endy, "Foundations For Engineering Biology," *Nature*, Vol. 438, pp. 449–453, 2005.

[5] M. Sedlak and N. Ho, "Production of Ethanol from Cellulosic Biomass Hydrolysate Using Genetically Engineered Yeast," *Applied Biochemistry & Biotechnology,* Vol. 114, No. 1-3, pp. 403–416, 2004.

[6] D.-K. Ro et al., "Production of the Antimalarial Drug Precursor Artemisinic Acid in Engineered Yeast," *Nature*, Vol. 440, pp. 940–943, 2006.

[7] J. Anderson, E. Clarke, A. Arkin, and C. Voigt, "Environmentally Controlled Invasion of Cancer Cells by Engineered Bacteria," *Journal of Molecular Biology*, Vol. 355, No. 4, pp. 619–627, 2006.

[8] D. Gillespie, "Exact Stochastic Simulation of Coupled Chemical Reactions," *Journal of Physical Chemistry*, Vol. 81, No. 25, pp. 2340–2361, 1977.

[9] M. Gibson and J. Bruck, "Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels," *Journal of Physical Chemistry A*, No. 104, pp. 1876–1889, 2000.

[10] M. Cook, "Networks of Relations," *Ph.D. Dissertation*, Advisor J. Bruck, Caltech, 2005.

[11] M. Samoilov, A. Arkin, and J. Ross, "Signal Processing by Simple Chemical Systems,"Journal of Physical Chemistry, Vol. 106, pp. 10205–10221, 2002.

[12] A. Arkin, J. Ross, and H. McAdams, "Stochastic Kinetic Analysis of Developmental Pathway Bifurcation in Phage λ-Infected E. Coli Cells," *Genetics*, Vol. 149, No. 1633, 1998.

[13] S. Paliwal, P. Iglesias, K. Hilioti, A. groisman, and A. Levchenko, "MAPK-mediated bimodal gene expression and adaptive gradient sensing in yeast", *Nature*, Vol. 446, Issue 7131, pp. 46–51, 2007.

[14] E. Libby, T. Perkins, and P. Swain, "Noisy information processing through transcriptional regulation," *Proceedings of the National Academy of Sciences*, Vol. 104, No. 17, pp. 7151–7156, 2007.

[15] BioBricks Parts List, *MIT Registry of Standard Biological Parts*,`http://parts.mit.edu`.

[16] I. Herskowitz, "Life Cycle of the Budding Yeast *Saccharomyces cerevisiae*," *Microbiological Reviews*, Vol. 52, No. 4, pp. 536-553, 1988.