Rate-Independent Constructs for Chemical Computation

Phillip Senum, Marc Riedel
Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, Minnesota, United States
E-mail: { senu0004, mriedel }@umn.edu

Abstract

This paper presents a collection of computational modules implemented with chemical reactions: an inverter, an incrementer, a decrementer, a copier, a comparator, a multiplier, an exponentiator, a raise-to-a-power operation, and a logarithm in base two. Unlike previous schemes for chemical computation, this method produces designs that are dependent only on coarse rate categories for the reactions ("fast" vs. "slow"). Given such categories, the computation is exact and independent of the specific reaction rates. The designs are validated through stochastic simulations of the chemical kinetics.

Introduction

The theory of reaction kinetics underpins our understanding of biological and chemical systems [1]. It is a simple and elegant formalism: chemical reactions define *rules* according to which reactants form products; each rule fires at a *rate* that is proportional to the quantities of the corresponding reactants that are present. On the computational front, there has been a wealth of research into efficient methods for simulating chemical reactions, ranging from ordinary differential equations (ODEs) [2] to stochastic simulation [3]. On the mathematical front, entirely new branches of theory have been developed to characterize the dynamics of chemical reaction networks [4].

Most of this work is from the vantage point of *analysis*: a set of chemical reaction exists, designed by nature and perhaps modified by human engineers; the objective is to understand and characterize its behavior. Comparatively little work has been done at a conceptual level in tackling the inverse problem of *synthesis*: how can one design a set of chemical reactions that implement specific behavior?

Of course, chemical engineers, genetic engineers and other practitioners strive to create novel functionality all the time. Generally, they begin with existing processes and pathways and modify these experimentally to achieve the desired new functionality [5, 6]. In a sense, much of the theoretical work on the dynamics of chemical reactions also addresses the synthesis problem by delineating the range of behaviors that are possible. For instance, theoretical work has shown that fascinating oscillatory and chaotic behaviors can occur in chemical reaction networks [7, 8].

Perhaps the most profound theoretical observation is that chemical reaction networks are, in fact, *computational processes*: regardless of the complexity of the dynamics or the subtlety of the timing, such networks transform *input* quantities of chemical species into *output* quantities through simple primitive operations. The question of the computational power of chemical reactions has been considered by several authors. Magnasco demonstrated that chemical reactions can compute anything that digital circuits can compute [9]. Soloveichik *et al.* demonstrated that chemical reactions are *Turing Universal*, meaning that they can compute anything that a computer algorithm can compute [10].

Such prior work considered the computational power of chemical reactions from a *deductive* point of view. This paper tackles the problem from an *inductive* point of view. We present a constructive method for designing specific computational modules: an inverter, an incrementer, a decrementer, a copier, a comparator, a multiplier, an exponentiator, a raise-to-a-power operation, and a logarithm in base two. This work builds upon our prior work that described constructs such as "for" and "while" loops [11] and signal processing operations such as filtering [12].

In contrast to previous work, our method produces designs that are dependent only on coarse rate categories for the reactions (e.g., "fast" and "slow"). It does not matter how fast any "fast" reaction is relative to another, or how slow any "slow" reaction is relative to another – only that "fast" reactions are fast relative to "slow" reactions. Specifically, suppose that we design a module that requires m slow reactions and n fast reactions. Any choice of m reactions with kinetic rate constants $k_1, \ldots k_m$ and n reactions with kinetic rate constants $k_{m+1}, \ldots, k_{m+n}$, where $k_i << k_j$, for all $i = 1, \ldots, m$, for all $j = m+1, \ldots, m+n$, will work.

The result of the computation is *rate-independent* in the sense that the *formula* of what is computed, say a logarithm, does not include any of the kinetic rate constants. We do not mean to imply that the rates do not matter. If the separation between "slow" and "fast" is not sufficiently large, then errors will occur. However, for a sufficiently large separation, the errors are small.

Indeed, the error that occurs as a function of the separation between "fast" and "slow" is our main criterion of goodness for our design. As Tables 1- 6 illustrate, our constructs perform remarkably well, computing with small errors for rate separations of 100 or 1,000 and vanishingly small errors for rate separations of 10,000. We validate our all of our designs through stochastic simulations of the chemical kinetics [13] using an open-source tool called Cain [14]. (Details about Cain can be found in the Appendix.)

Chemical Model

We adopt the model of discrete, stochastic chemical kinetics [3, 15]. Molecular quantities are whole numbers (i.e., non-negative integers). Reactions fire and alter these quantities by integer amounts. The reaction rates are proportional to (1) the quantities of the reacting molecular types; and (2) kinetic rate constants. As discussed above, all of our designs are formulated in terms of two coarse kinetic rate constant categories ("fast" and "slow").

Consider the reaction

$$X_1 \rightarrow X_2 + X_3. \tag{1}$$

When this reaction fires, one molecule of X_1 is consumed, one of X_2 is produced, and one of X_3 is produced. (Accordingly, X_1 is called a *reactant* and X_2 and X_3 the *products*.) Consider what this reaction accomplishes from a computational standpoint. Suppose that it fires until all molecules of X_1 have been consumed. This results in quantities of X_2 and X_3 equal to the original quantity of X_1, and a new quantity of X_1 equal to zero:

```
X2 := X1
X3 := X1
X1 := 0
```

Consider the reaction

$$X_1 + X_2 \rightarrow X_3. \tag{2}$$

Suppose that it fires until either all molecules of X_1 or all molecules of X_2 have been consumed. This results in a quantity of X_3 equal to the lesser of the two original quantities:

```
X3 := min(X1, X2)
X1 := X1 - min(X1, X2)
X2 := X2 - min(X1, X2)
```

We will present constructs different arithmetical and logical operations in this vein. Each sets the final quantity of some molecular type as a function of the initial quantities of other types.

Most of our designs consist of either unimolecular or bimolecular reactions, i.e., reactions with one or two reactants, respectively. A small subset of the reactions are trimolecular. Mapping these to chemical substrates might not be feasible, since the kinetics of reactions with more than two reactants are complex and often physically unrealistic. For all trimolecular reactions, we suggest the follow generic scheme for converting them into bimolecular reactions. (This idea is found in [16] in the context of DNA strand displacement reactions.) We convert any trimolecular reaction

$$a + b + c \rightarrow d \tag{3}$$

into a pair of reactions

$$\begin{aligned} a + b &\rightleftharpoons e &(4)\\ e + c &\rightarrow d &(5) \end{aligned}$$

where e is an new intermediary type. Note that Reaction 4 is a reversible reaction. We assume that this reaction is fast relative to all others. Accordingly, if there are non-zero quantities of a and b but zero of c, the system will “back-off”, converting e back into a and b. Other reactions in the system that use a and b can continue to fire.

Apart from reactions resulting from such trimolecular conversions, we do not use reversible reactions in any of our constructs. Of course, all chemical reactions are reversible. Implicitly, we assume that all reverse rates are much slower that the forward reactions (except for those corresponding to Reaction 4).

Computational Constructs

In this section, we present a collection of constituent constructs for rate-independent computation: an inverter, an incrementer/decrementer, a copier, and a comparator. In the next section, we use some of these constructs to implement a multiplier, a logarithm operation, an exponentiator, and a raise-to-the-power operation. A reference of all reactions needed for these constructs can be found in the Appendices.

An Inverter

We implement an operation that is analogous to that performed by an inverter (i.e., a NOT gate) in a digital system: given a non-zero quantity (corresponding to logical “1”) we produce a zero quantity (corresponding to logical “0”). Conversely, given a zero quantity, we produce a non-zero quantity. We accomplish this with a pair of chemical types: the given type, for example, a, and a corresponding “**absence indicator**” type, which will be referred to as a_{ab}. The reactions generating the absence indicator are shown in reactions 6–8.

$$\begin{aligned} \varnothing &\xrightarrow{\text{slow}} a_{\mathrm{ab}} &(6)\\ a + a_{\mathrm{ab}} &\xrightarrow{\text{fast}} a &(7)\\ 2\,a_{\mathrm{ab}} &\xrightarrow{\text{fast}} a_{\mathrm{ab}} &(8) \end{aligned}$$

Note that when the empty set symbol, $\varnothing$, is used as a reactant, it indicates that the reactants are a large or replenishable unreactive source; when it is used as a product, it indicates that the products of the reaction are waste.

The first reaction continuously generates molecules of a_{ab}, so in the absence of molecules of a we will have a non-zero quantity of a_{ab} in the system. If there are molecules of a present, then second reaction quickly consumes any molecules of a_{ab} that are generated, so the quantity of a_{ab} will be close to zero. The third reaction ensures that the quantity a_{ab} remains small.

We use this simple construct in many of our computational modules [12,17]. It is also a fundamental part of all of the constructs introduced in this paper. In general, it can be used to synchronize steps.

Suppose that we want to perform an operation similar to the one in reactions 9–10.

$$a \rightarrow b \tag{9}$$
$$b \rightarrow [\textit{operate on } b] \tag{10}$$

Here the second step is an operation that depends on the quantity of b. We do not want to start consuming molecules of b until the full quantity of it is generated from a. We can accomplish this with an absence indicator a_{ab}:

$$a \rightarrow b \tag{11}$$
$$a_{\text{ab}} + b \rightarrow [\textit{operate on } b] \tag{12}$$

It is important to note that absence indicators generated by reactions 6–8 can only be used with "slow" reactions. If they were used by a "fast" reaction, it is possible that a false positive could be detected because a "fast" will compete with reaction 7. In situations where absence indicators need to be consumed by "fast" reactions, we can use an alternate two-step process to produce them.

$$\varnothing \xrightarrow{\text{slow}} a_{\text{ab}} \tag{13}$$
$$a + a_{\text{ab}} \xrightarrow{\text{fast}} a \tag{14}$$
$$2\, a_{\text{ab}} \xrightarrow{\text{fast}} a_{\text{ab}} \tag{15}$$
$$a_{\text{ab}} \xrightarrow{\text{slow}} a'_{\text{ab}} \tag{16}$$
$$a + a'_{\text{ab}} \xrightarrow{\text{fast}} a \tag{17}$$
$$2\, a'_{\text{ab}} \xrightarrow{\text{fast}} a'_{\text{ab}} \tag{18}$$

In this case, a secondary absence indicator, a'_{ab} is produced from a_{ab} through a "slow" reaction. This allows "fast" reactions to use a'_{ab} safely because it is buffered through reaction 16.

Increment and Decrement Operations

We describe constructs to implement incrementation and decrementation. These operations form the basis of more complex arithmetical operations, such as multiplication. The inputs consist of two molecular types: g, the "start signal;" and x, the quantity to be incremented or decremented. We assume that some external source injects molecules of g. Any quantity can be injected; regardless, the quantity of x is incremented or decremented by exactly one, consuming all the molecules g in the process. The operations proceed as follows:

1) The system waits for the start signal g to be some non-zero quantity.

2) It transfers the quantity of x to a temporary type x'.

3) It sets g to zero.

4) It transfers all but one molecule of x' back to x.

5a) For a decrement, it removes the last molecule x'.

5b) For an increment, it removes the last molecule of x' and adds to two molecules to x.

The following reactions implement this scheme. Given molecules of g, a reaction transfers molecules of x to molecules of x':

$$x + g \xrightarrow{\text{slow}} x' + g \tag{19}$$

The following reaction sets the quantity of g to zero. Using the absence indicator mechanism described in the preceding section, it does so only once all molecules of x have been transfered to x':

$$g + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{20}$$

Reactions of the form of 6–8 are needed to generate x_{ab}; we omit them here. The following reaction transfers all but one molecule of x' back to x. It does so by repeatedly selecting pairs of x' and turning one molecule of x' into x. In essence, this is a repeated integer division by two. Again, using the absence indicator mechanism, it proceeds only once all molecules of g have been removed:

$$g'_{\text{ab}} + 2\,x' \xrightarrow{\text{fast}} x + x' + x^{\text{rx}} \tag{21}$$

In reaction 21, we do not directly use an absence indicator for g_{ab}, but instead, we use a secondary absence indicator g'_{ab}, generated in the method outlined in reactions 13–18.

Reaction 21 also produces molecules of a supplementary type x^{rx}. Note that this reaction is in the "fast" category. The new type x^{rx} is consumed by the reaction:

$$x^{\text{rx}} \xrightarrow{\text{slow}} \varnothing. \tag{22}$$

Note that this reaction is in the "slow" category. We introduce x^{rx} because we cannot directly use an absence indicator for x' to detect when reaction 21 has completed because x' is never completely consumed. Instead, we use x^{rx} to indicate that we are currently transferring molecules of x' back to x; it is consumed when the step completes. Again, reactions of the same form as 6– 8 are needed to generate $x^{\text{rx}}_{\text{ab}}$; we omit them here.

Finally, we include the following reaction to perform a decrement:

$$x^{\text{rx}}_{\text{ab}} + x' + g'_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \quad \text{[Decrement]} \tag{23}$$

Or we include the following reaction to perform an increment:

$$x^{\text{rx}}_{\text{ab}} + x' + g'_{\text{ab}} \xrightarrow{\text{slow}} 2\,x \quad \text{[Increment]} \tag{24}$$

With a slight modification of reaction 21, we can also implement division by 2 with this module:

$$g'_{\text{ab}} + 2\,x' \xrightarrow{\text{fast}} x + x^{\text{rx}} \tag{25}$$

A Copier

In digital computation, one of the most basic operations is copying a quantity from one register into another. The programming construct is "set the value of b to be the value of a":

```
let b := a;
```

To implement an equivalent operation with chemical reactions, we could use a reaction that simply transfers the quantity of a to b:

$$a \rightarrow b \tag{26}$$

However, this is not ideal because this reaction consumes all the molecules of a, setting its quantity to zero. We would like a chemical construct that copies the quantity without altering it. The following reaction does not work either:

$$a \rightarrow a + b \tag{27}$$

It just creates more and more molecules of b in the presence of a. A more sophisticated construct is needed.

In our construct, we have a "start signal" type g. When an external source injects molecules of g, the copy operation proceeds. (In the same way as our increment and decrement operations, the quantity of g that is injected is irrelevant.) It produces an output quantity of b equal to the input quantity of a; it leaves the quantity of a unchanged. The reactions for the copier construct are as follows. Firstly, in the presence of g, a reaction transfers the quantity of a to a':

$$g + a \xrightarrow{\text{slow}} g + a' \tag{28}$$

Secondly, after all molecules of a have been transferred to a', the system removes all the molecules of g:

$$g + a_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{29}$$

Here, again, we are using the concept of an absence indicator. Removing g ensures that a is copied exactly once. After g has been removed, a reaction transfers the quantity of a' back to a and also creates this same quantity of b:

$$g_{\text{ab}} + a' \xrightarrow{\text{slow}} a + b \tag{30}$$

Alternatively, we can use a slight modification of this reaction to double the quantity of a:

$$g_{\text{ab}} + a' \xrightarrow{\text{slow}} 2\,a \tag{31}$$

We also generate absence indicators a_{ab} and g_{ab} by the same method as reactions 6–8. We note that, while this construct leaves the quantity of a unchanged after it has finished executing, it temporarily consumes molecules a, transferring the quantity of these to a' before transferring it back. Accordingly, no other constructs should use a in the interim.

A Comparator

Using our copier construct, we can create a construct that compares the quantities of two input types and produces an output type if one is greater than the other. For example, let us assume that we want to compare the quantities of two types a and b:

```
if  (a > b) {
    t := TRUE
} else {
    t := FALSE
}
```

If the quantity of a is greater than the quantity of b, the system should produce molecules of an output type t; otherwise, it should not produce any molecules of t.

First, we create temporary copies, a^c and b^c, of the types that we wish to compare, a and b, using the copier construct described in the previous section. (We omit these reactions; they are two verbatim copies of the copier construct, one with a as an input and a^c as an output, the other with b as an input and b^c as an output.) We split the start signal so that the two copiers are not competing for it:

$$g \xrightarrow{\text{fast}} g^1 + g^2 \tag{32}$$

Now we compare a and b via their respective copies a^c and b^c. To start, we first consume pairs of a^c and b^c:

$$a^c + b^c \rightarrow \varnothing \tag{33}$$

We assume that this reaction fires to completion. The result is that there are only molecules of a^c left, or only molecules of b^c left, or no molecules of a^c nor b^c left. Molecules of the type that originally had a larger quantity have persisted. If the quantities were equal, then both types were annihilated. We use absence indicators a^c_{ab} and b^c_{ab} to determine which type was annihilated, produced by the method shown in reactions 6–8. If a was originally greater than b, there will now be a presence of a^c and an absence of b^c. We produce molecules of type t if this condition is met. We preserve the quantities of a^c and b^c_{ab}. We can also limit the quantity of t produced by introducing a fuel type:

$$\text{fuel} + a^c + b^c_{\text{ab}} \xrightarrow{\text{slow}} a^c + b^c_{\text{ab}} + t \tag{34}$$

For robustness, we also add reactions to destroy t in the case that the asserted condition is not true:

$$a^c_{\text{ab}} + b^c + t \xrightarrow{\text{slow}} a^c_{\text{ab}} + b^c \tag{35}$$

$$a^c_{\text{ab}} + b^c_{\text{ab}} + t \xrightarrow{\text{slow}} a^c_{\text{ab}} + b^c_{\text{ab}} \tag{36}$$

We can readily generalize the construct to all types of logical comparisons. Table 1 lists these operations and their corresponding reactions.

Complex Arithmetic

Based upon the modules described in the previous section, we provide examples of how to implement more complex arithmetic: multiplication, logarithms in base two, exponentiation, and raising to a power. In order to elucidate the designs, we specify the sequence of operations for each of these module in pseudo-code. The pseudo-code operations consist of:

- Assignment, addition, and subtraction operations. The operands may be constants or variables.
- Decision-making constructs: `while` and `if` statements. The logical test for each of these constructs can be any one of the six conditions listed in Table I. In some cases, the `if` and `while` statements will be nested.

A Multiplier

Building upon the constructs in the last section, we show a construct that multiplies the quantities of two input types. Multiplication can be implemented via iterative addition. Consider the following lines of pseudo-code:

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

The result is that z is equal to x times y. We implement multiplication chemically using the constructs described in the previous sections: the line `z = z + y` is implemented with a copy operation; the line `x = x - 1` is implemented using a decrement operation. A third set of reactions handle the looping behavior of the `while` statement.

Firstly, we have reactions that copy the quantity of y to z. We use "start signal" types g^1 and g^2 to synchronize iterations; it is supplied from the controlling reaction 48 below.

$$g^1 + y \xrightarrow{\text{slow}} g^1 + y' \tag{37}$$

$$g^1 + y_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{38}$$

$$g^1_{\text{ab}} + y' \xrightarrow{\text{slow}} y + z \tag{39}$$

Secondly, we have reactions that decrement the value of x. We use g^2 as the signal to begin the decrement.

$$x + g^2 \xrightarrow{\text{slow}} x' + g^2 \tag{40}$$

$$g^2 + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{41}$$

$$2\,x' + g'^2_{\text{ab}} \xrightarrow{\text{fast}} x' + x + x^{\text{rx}} \tag{42}$$

$$x^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{43}$$

$$x' + x^{\text{rx}}_{\text{ab}} + g'^2_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{44}$$

Thirdly, we have a controlling set of reactions to implement the `while` statement. This set generates g^1 and g^2 to begin the next iteration, preserving the quantity of x:

$$x + x'_{\text{ab}} + y'_{\text{ab}} \xrightarrow{\text{slow}} x + g^P \tag{45}$$

$$g^P + x' \xrightarrow{\text{fast}} x' \tag{46}$$

$$g^P + y' \xrightarrow{\text{fast}} y' \tag{47}$$

$$g^P \xrightarrow{\text{slow}} g^1 + g^2 \tag{48}$$

This set initiates the next iteration of the loop if such an iteration is not already in progress and if there are still molecules of x in the system. The types x' and y' are present when we are decrementing x or copying y, respectively; thus, they can be used to decide whether we are currently inside the loop or not. Finally, we generate the four absence indicators according to the template in reactions 6– 8.

Raise to a Power

As a second complex example, we show how to implement the operation $y = x^p$. This can be done using iterative multiplication; as we demonstrated in the previous section, multiplication can be implemented via iterative addition. The pseudo-code for the raising-to-a-power operation is shown in Figure 1. It consists of assignment, addition, subtraction, and iterative constructs. Note that the assignment operations can be performed with our "copier" module; the addition and subtraction operations can be performed with "increment" and "decrement" modules. A pair of nested `while` constructs, similar to that used for multiplication, perform the requisite iterative computation. The complete set of reactions to implement this operation is given in the Appendix.

Exponentiation

To implement the operation $y = 2^x$, we can use a sequence of operations similar to those that we used for multiplication. The pseudo-code is shown in Figure 2. The reactions that implement this pseudo-code are given in the Appendix.

Logarithm

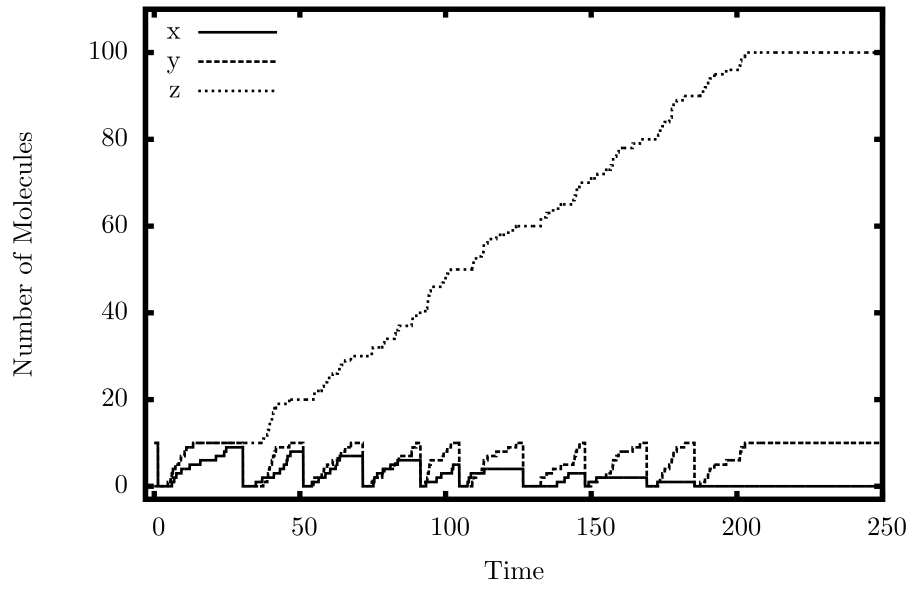
We demonstrate the computation of a base-2 logarithm. The pseudo-code is shown in Figure 3. A logarithm is the inverse operation of exponentiation; it makes sense, therefore, that the pseudo-code for log_2 is more or less the reverse of that for exponentiation. The reactions that implement this operation are given in the Appendix.

Simulation Results

We validated our constructs using stochastic simulation. Specifically, we performed a time homogeneous simulation using Gillespie's "Direct Method" [3] with the software package "Cain" from Caltech [14]. (Details about Cain can bew found in the Appendix.) In each case, the simulation was run until the quantities of all types except the absence indicators converged to a steady state. We used a rate constant of 1 for the "slow" reactions. We tried rate constants between two to four orders of magnitude higher for the "fast" reactions. (We refer to the ratio of "fast" to "slow" as the *rate separation.*) For each of the graphs below, the initial quantity of each type is zero, with the exception of the types specified.

Multiplier

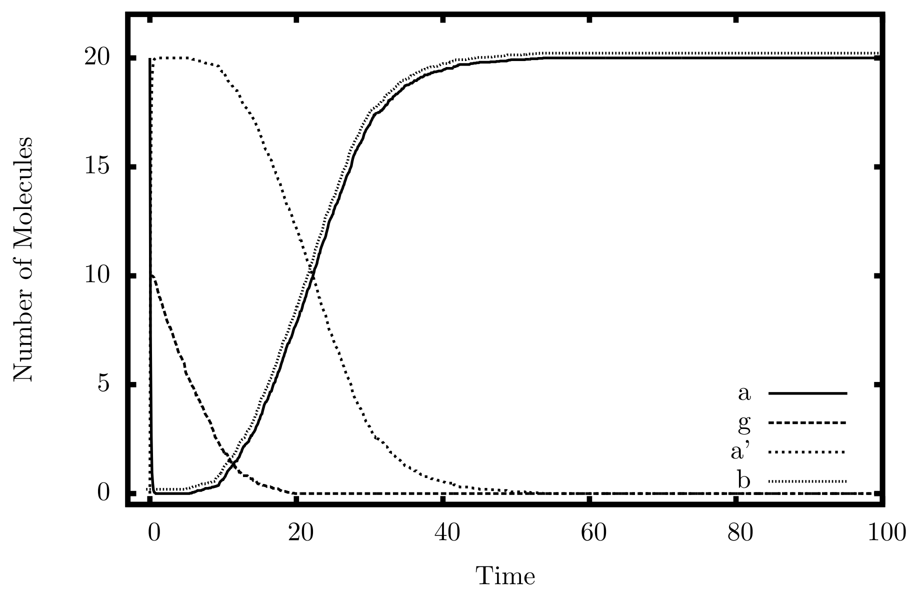
Graph 1: Simulated Multiplier, $x = 10$, $y = 10$



Graph 1 shows the output of a single simulated trajectory for our multiplier. We observe exactly the behavior that we are looking for: the quantity of y cycles exactly 10 times as it exchanges with y' and is copied to z; the quantity of z grows steadily up to 100; the quantity of x decreases once each cycle down to 0. Table 2 presents detailed simulation results, this time tested for accuracy. Errors generally occur if the system executes too many or too few iterations. As can be seen, the larger the quantity of x, the more accurate the result, in relative terms. As expected, the larger the rate separation, the fewer errors we get.

Copier

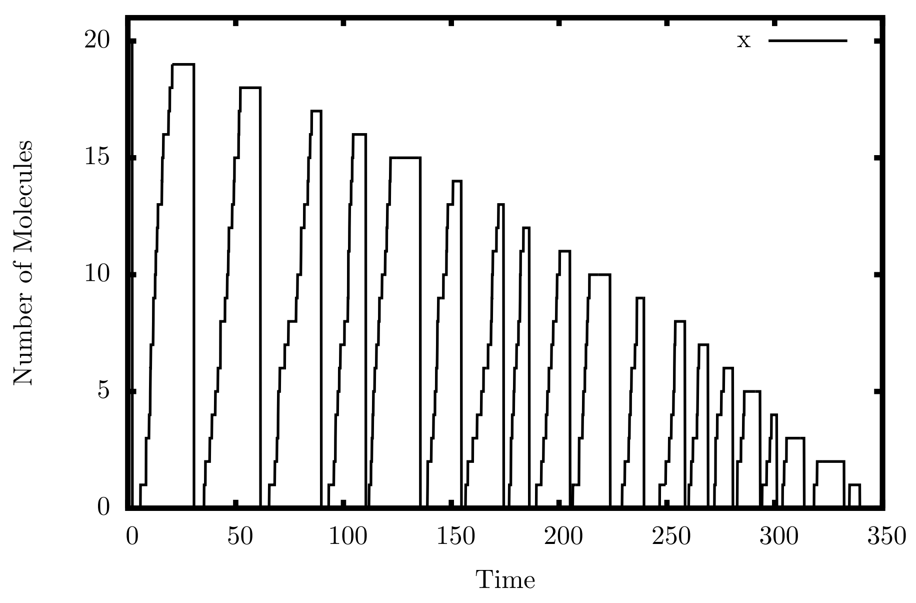
Graph 2: Simulated Copier, $a = 20$, $cr = 10$



Graph 2 shows an average simulated trajectory for our copier. Again, we observe exactly the behavior we expect: the quantity of a drops to 0 almost immediately as it turns into a'; this is followed by the removal of g from the system. When the quantity of g drops to nearly zero, both a and b rise steadily back to the original quantity of a. Table 3 shows additional simulation results from our copier, this time tested for accuracy. The copier construct appears to be quite robust to errors; however, large rate separations do not help as much as they do for the multiplier. The system seems to prefer a larger injection quantity of g, but whether it is larger or smaller than the initial quantity of a is irrelevant.

Decrementer

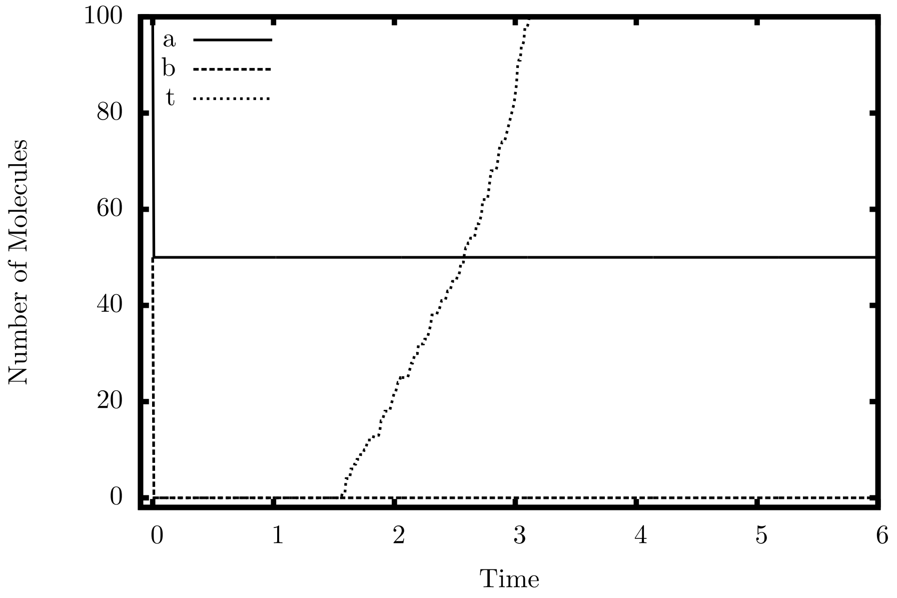
Graph 3: Simulated Decrement, $x = 20$



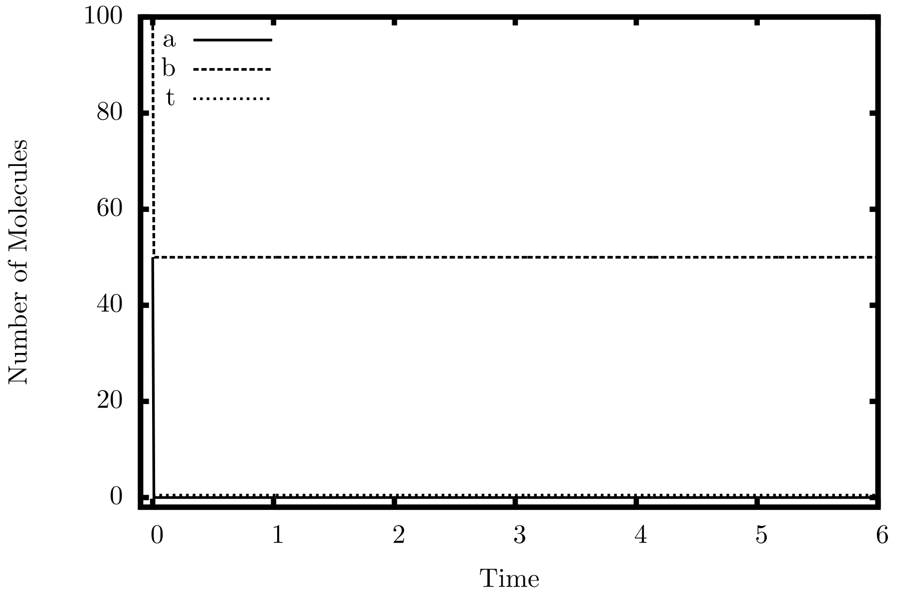
Graph 3 shows the output of a single simulated trajectory of our decrementer. An automatic restart mechanism, similar to reactions 45–48, was used to produce a continuous series of decrements. Exactly twenty peaks can be seen in the graph, including the initial peak on the far-left margin of the graph. This is exactly the behavior we are looking for – a decrement by exactly one each cycle.

Comparator

Graph 4: Comparator ($a > b$), $a = 100$, $b = 50$



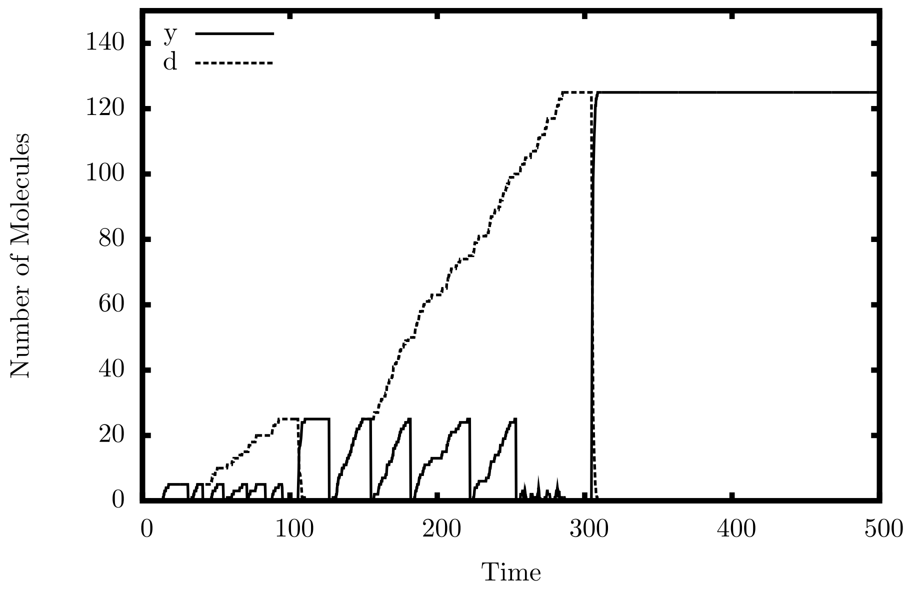
Graph 5: Comparator ($a > b$), $a = 50$, $b = 100$



Graphs 4 and 5 display simulation results from our comparator. In Graph 4, t is asserted as we would expect; in Graph 5, t is not asserted, also as we would expect.

Raise to a Power

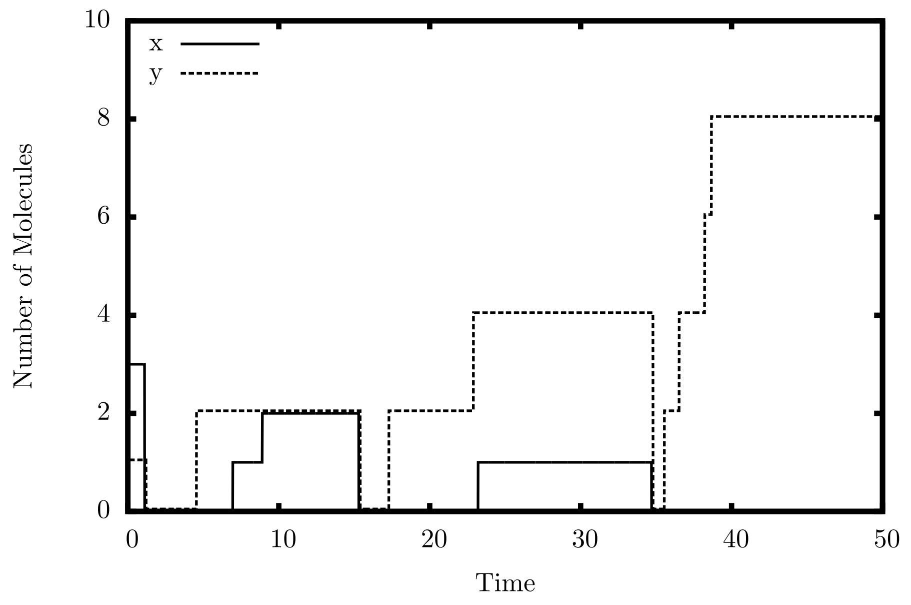
Graph 6: Simulated Raise to Power, $x = 5$, $p = 3$, $begin = 10$



Graph 6 shows a simulated trajectory of our raise-to-a-power construct. As can be observed, after y is loaded with the initial quantity of x, it is multiplied by x twice. Each time its value is stored in the temporary type d before being transferred back. Table 4 shows simulation results for our raise-to-a-power construct for various values of x and p. In each case, the initial quantity of *begin* was set to 10, simulating an injection of that type.

Exponentiation

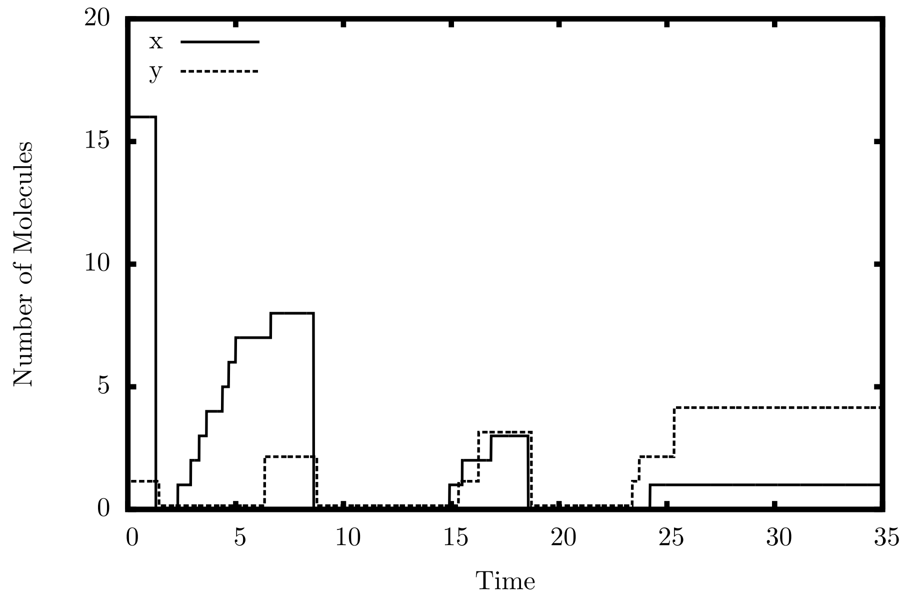
Graph 7: Simulated Exponentiation, $x = 3$, $y = 1$



Graph 7 shows a simulated trajectory of our exponentiation construct. We can observe that for every decrementation of x, y doubles in value, which is the behavior that we are aiming for. Table 5 shows more simulation results. The error for this construct is small but appears to grow as x grows. This is not surprising, given we are performing exponentiation: small errors will be compounded.

Logarithm

Graph 8: Simulated Logarithm, $x = 16$, $y = 1$



Graph 8 shows a simulated trajectory for our base-2 logarithm construct. Again, we observe the behavior that we are expecting; every time we divide x by two, y increases by one. Table 6 shows more detailed simulation results.

Discussion

This paper presented a collection of specific computational constructs. More complex operations – multiplication, exponentiation, raising to a power, and logarithms – were built a collection of robust, primitive operations – absence indicators, incrementing and decrementing, copying, and comparing. The process by which we assembled these primitive operations could be readily generalized. Indeed, we are developing a chemical compiler that will translate any sequence of operations specified by pseudo-code into chemical reactions. The compiler will accept general pseudo-code written in the vein of that shown in Figures 1–3. It will allow for assignments, arithmetic operations, "if" statements, and arbitrarily nested "while" loops.

The novelty and value of the constructs that we have demonstrated is that they are all rate independent. Here "rate independent" refers to the fact that, within a broad range of values for the kinetic constants, the computation does not depend on the specific values of the constants. Of course, outside of this range, the accuracy of the computation degrades. For rates within the target range, our results are remarkably accurate: in nearly all cases the errors are less than 1%. In many cases, the errors are much less than 1%. The actual value of the target range will depend on the chemical substrate used; in simulation, it was found that a ratio of 10,000:1 of "fast" vs. "slow" produced nearly perfect results.

Our contribution is to tackle the problem of synthesizing computation at a conceptual level, working not with actual molecular types but rather with abstract types. One might question whether actual chemical reactions matching our templates can be found. Certainly, engineering complex new reaction mechanisms in any experimental domain is formidable task; for *in vivo* systems, there are likely to be many experimental constraints on the choice of reactions [18]. However, we point to recent work on *in vitro* computation as a potential application domain for our ideas.

Through a mechanism called DNA strand-displacement, a group at Caltech has shown that DNA reactions can emulate the chemical kinetics of nearly any chemical reaction network. They also provide a compiler that translates abstract chemical reactions of the sort that we design into specific DNA reactions [16]. Recent work has demonstrated both the scale of computation that is possible with DNA-based computing [19], as well as exciting applications [20]. While conceptual, our work suggest a *de novo* approach to the design of biological functions. Potentially this approach is more general in its applicability than methods based on appropriating and reusing existing biological modules.

Acknowledgments

This research is supported by an NSF CAREER Award, #0845650, and by a UROP Award from the University of Minnesota.

References

1. Horn F, Jackson R (1972) General mass action kinetics. Archive for Rational Mechanics and Analysis 47: 81–116.

2. Érdi P, Tóth J (1989) Mathematical Models of Chemical Reactions: Theory and Applications of Deterministic and Stochastic Models. Manchester University Press.

3. Gillespie D (1977) Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry 81: 2340-2361.

4. Strogatz S (1994) Nonlinear Dynamics and Chaos with Applications to Physics, Biology, Chemistry, and Engineering. Perseus Books.

5. Win MN, Liang J, Smolke CD (2009) Frameworks for programming biological function through RNA parts and devices. Chemistry & Biology 16: 298–310.

6. Keasling J (2008) Synthetic biology for synthetic chemistry. ACS Chemical Biology 3: 64–76.

7. Epstein IR, Pojman JA (1998) An Introduction to Nonlinear Chemical Dynamics: Oscillations, Waves, Patterns, and Chaos. Oxford Univ Press.

8. Willamowski KD, Rössler OE (1980) Irregular oscillations in a realistic abstract quadratic mass action system. Zeitschrift fur Naturforschung Section A – A Journal of Physical Sciences 35: 317-318.

9. Magnasco MO (1997) Chemical kinetics is turing universal. Phys Rev Lett 78: 1190–1193.

10. Soloveichik D, Cook M, Winfree E, Bruck J (2008) Computation with finite stochastic chemical reaction networks. Natural Computing 7.

11. Shea A, Fett B, Riedel MD, Parhi K (2010) Writing and compiling code into biochemistry. In: Proceedings of the Pacific Symposium on Biocomputing. pp. 456–464.

12. Jiang H, Kharam AP, Riedel MD, Parhi KK (2010) A synthesis flow for digital signal processing with biomolecular reactions. In: IEEE International Conference on Computer-Aided Design. pp. 417–424.

13. Gillespie DT (1976) A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. Journal of Computational Physics 22: 403–434.

14. Mauch S, Stalzer M (2009) Efficient formulations for exact stochastic simulation of chemical systems. IEEE/ACM Transactions on Computational Biology and Bioinformatics 99.

15. Gillespie DT (2006) Stochastic simulation of chemical kinetics. Annual Review of Physical Chemistry 58: 35–55.

16. Soloveichik D, Seelig G, Winfree E (2010) DNA as a universal substrate for chemical kinetics. Proceedings of the National Academy of Sciences 107: 5393–5398.

17. Kharam A, Jiang H, Riedel MD, Parhi K (2011) Binary counting with chemical reactions. In: Pacific Symposium on Biocomputing.

18. Weiss R (2003) Cellular Computation and Communications using Engineering Genetic Regulatory Networks. Ph.D. thesis, MIT.

19. Qian L, Winfree E (2009) A simple DNA gate motif for synthesizing large-scale circuits. In: DNA Computing. pp. 70-89.

20. Venkataramana S, Dirks RM, Ueda CT, Pierce NA (2010 (in press)) Selective cell death mediated by small conditional RNAs. Proceedings of the National Academy of Sciences .

Tables

Table 1. Logical operations via chemical reactions.

Operation	Creation	Destruction	Operation	Creation	Destruction
`a == b`	$a_{\mathrm{ab}} + b_{\mathrm{ab}}$	$a + b_{\mathrm{ab}}$	`a >= b`	$a + b_{\mathrm{ab}}$	$a_{\mathrm{ab}} + b$
		$a_{\mathrm{ab}} + b$		$a_{\mathrm{ab}} + b_{\mathrm{ab}}$	
`a > b`	$a + b_{\mathrm{ab}}$	$a_{\mathrm{ab}} + b$	`a <= b`	$a_{\mathrm{ab}} + b$	$a + b_{\mathrm{ab}}$
		$a_{\mathrm{ab}} + b_{\mathrm{ab}}$		$a_{\mathrm{ab}} + b_{\mathrm{ab}}$	
`a < b`	$a_{\mathrm{ab}} + b$	$a + b_{\mathrm{ab}}$	`a != b`	$a_{\mathrm{ab}} + b$	$a_{\mathrm{ab}} + b_{\mathrm{ab}}$
		$a_{\mathrm{ab}} + b_{\mathrm{ab}}$		$a + b_{\mathrm{ab}}$	

Table 2. Statistical simulation results for "Multiplier" construct.

Trial	Rate Separation	Trajectories	x	y	z	Expected z	Error
1	100	100	100	50	4954.35	5000	0.91%
2	100	100	50	100	4893.18	5000	2.14%
3	1000	100	100	50	4991.56	5000	0.17%
4	1000	100	50	100	4995.78	5000	0.08%
5	10000	100	100	50	4998.69	5000	$< 0.01\%$
6	10000	100	50	100	4999.14	5000	$< 0.01\%$
7	10000	100	10	20	200.04	200	$< 0.01\%$
8	10000	100	20	10	200.03	200	$< 0.01\%$

Table 3. Statistical simulation results for "Copier" construct.

Trial	Rate Separation	Trajectories	cr	a	b	Expected b	Error
1	100	500	5	100	102.45	100	2.45 %
2	100	500	50	100	104.826	100	4.826%
3	1000	500	5	100	100.312	100	0.312%
4	1000	500	50	100	100.516	100	0.516%
5	10000	500	5	100	100.022	100	0.022%
6	10000	500	50	100	100.034	100	0.034%
7	10000	500	5	5000	4938.39	5000	1.232%
8	10000	500	50	5000	4967.26	5000	0.655%
9	10000	500	200	5000	4796.38	5000	4.072%
10	10000	500	50	2	2	2	<0.01 %

Table 4. Statistical Simulation Results from "Raise to a Power" Construct.

Trial	Rate Separation	Trajectories	x	p	y	Expected y	Error
1	10000	100	3	9	19734.3	19683	0.26%
2	10000	100	4	8	64884.7	65536	0.99%
3	10000	100	5	4	626.87	625	0.30%
4	10000	100	6	7	279864	279936	0.03%
5	10000	100	9	6	531412	531441	<0.01%
6	10000	100	10	3	999.43	1000	0.06%

Table 5. Statistical Simulation Results from "Exponentiation" Construct.

Trial	Rate Separation	Trajectories	x	y	Expected y	Error
1	10000	100	2	4	4	<0.01%
2	10000	100	3	8	8	<0.01%
3	10000	100	6	64.32	64	0.50%
4	10000	100	9	514.3	512	0.45%
5	10000	100	11	2051.48	2048	0.17%
6	10000	100	19	523461	524228	0.15%

Table 6. Statistical Simulation Results from "Logarithm" Construct.

Trial	Rate Separation	Trajectories	x	y_f	Expected y_f	Error
1	10000	100	2	1	1	<0.01%
2	10000	100	10	3	3	<0.01%
3	10000	100	62	5	5	<0.01%
4	10000	100	83	6	6	<0.01%
5	10000	100	163	7	7	<0.01%
6	10000	100	286	7.99	8	<0.01%
7	10000	100	1165	10	10	<0.01%

Appendix: Module Reference

Absence Indicator

Generates a_{ab} in the absence of a, and consumes a_{ab} in the presence of a without modifying a.

Reactions

$$\varnothing \xrightarrow{\text{slow}} a_{\text{ab}} \tag{49}$$

$$a + a_{\text{ab}} \xrightarrow{\text{fast}} a_{\text{ab}} \tag{50}$$

$$2\,a_{\text{ab}} \xrightarrow{\text{fast}} a_{\text{ab}} \tag{51}$$

Copier

Adds the value stored in y to z in the presence of g, preserving the initial quantity of y.

Reactions

$$y + g \xrightarrow{\text{slow}} y' + g \tag{52}$$

$$g + y_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{53}$$

$$g_{\text{ab}} + y' \xrightarrow{\text{slow}} y + z \tag{54}$$

Absence Indicators

$$\varnothing \xrightarrow{\text{slow}} y_{\text{ab}} \tag{55}$$

$$y + y_{\text{ab}} \xrightarrow{\text{fast}} y_{\text{ab}} \tag{56}$$

$$2\,y_{\text{ab}} \xrightarrow{\text{fast}} y_{\text{ab}} \tag{57}$$

$$\varnothing \xrightarrow{\text{slow}} g_{\text{ab}} \tag{58}$$

$$g + g_{\text{ab}} \xrightarrow{\text{fast}} g \tag{59}$$

$$2\,g_{\text{ab}} \xrightarrow{\text{fast}} g_{\text{ab}} \tag{60}$$

Multiply by Two

Takes the value stored in y and doubles it in the presence of g, putting the result back in y.

Reactions

$$y + cr \xrightarrow{\text{slow}} y' + cr \tag{61}$$

$$g + y_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{62}$$

$$g_{\text{ab}} + y' \xrightarrow{\text{slow}} 2\,y \tag{63}$$

Absence Indicators Same as for “Copier.”

Decrement

Takes the values of x and subtracts 1 from it in the presence of g, putting the result back in x.

Reactions

$$x + g \xrightarrow{\text{slow}} x' + g \tag{64}$$

$$g + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{65}$$

$$2\,x' + g'_{\text{ab}} \xrightarrow{\text{fast}} x + x' + x^{\text{rx}} \tag{66}$$

$$x^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{67}$$

$$x' + x^{\text{rx}}_{\text{ab}} + g'_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{68}$$

Absence Indicators

$$g_{\text{ab}} \xrightarrow{\text{slow}} g'_{\text{ab}} \tag{69}$$

$$g + g'_{\text{ab}} \xrightarrow{\text{fast}} g \tag{70}$$

$$2\,g'_{\text{ab}} \xrightarrow{\text{fast}} g'_{\text{ab}} \tag{71}$$

$$\varnothing \xrightarrow{\text{slow}} x_{\text{ab}} \tag{72}$$

$$x + x_{\text{ab}} \xrightarrow{\text{fast}} x \tag{73}$$

$$2\,x_{\text{ab}} \xrightarrow{\text{fast}} x_{\text{ab}} \tag{74}$$

$$\varnothing \xrightarrow{\text{slow}} x^{\text{rx}}_{\text{ab}} \tag{75}$$

$$x^{\text{rx}} + x^{\text{rx}}_{\text{ab}} \xrightarrow{\text{fast}} x^{\text{rx}} \tag{76}$$

$$2\,x^{\text{rx}}_{\text{ab}} \xrightarrow{\text{fast}} x^{\text{rx}}_{\text{ab}} \tag{77}$$

$$\varnothing \xrightarrow{\text{slow}} g_{\text{ab}} \tag{78}$$

$$g + g_{\text{ab}} \xrightarrow{\text{fast}} g \tag{79}$$

$$2\,g_{\text{ab}} \xrightarrow{\text{fast}} g_{\text{ab}} \tag{80}$$

Increment

Takes the values of x and adds 1 to it in the presence of g, putting the result back in x.

Reactions

$$x + g \xrightarrow{\text{slow}} x' + g \tag{81}$$

$$g + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{82}$$

$$2\,x' + g'_{\text{ab}} \xrightarrow{\text{fast}} x + x' + x^{\text{rx}} \tag{83}$$

$$x^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{84}$$

$$x' + x^{\text{rx}}_{\text{ab}} + g'_{\text{ab}} \xrightarrow{\text{slow}} 2\,x \tag{85}$$

Absence Indicators Same as for "Decrement."

Divide by Two

Takes the values of x and halves it in the presence of g, putting the result back in x.

Reactions

$$x + g \xrightarrow{\text{slow}} x' + g \tag{86}$$

$$g + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{87}$$

$$2\,x' + g'_{\text{ab}} \xrightarrow{\text{fast}} x + x^{\text{rx}} \tag{88}$$

$$x^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{89}$$

$$x' + x^{\text{rx}}_{\text{ab}} + g'_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{90}$$

Absence Indicators Same as for "Decrement."

Appendix: Multiplication Reactions

System Initialization We use a small set of reactions to implement the iterative loop of the operation.

$$x + x'_{\text{ab}} + y'_{\text{ab}} \xrightarrow{\text{slow}} x + g^P \tag{91}$$
$$g^P + x' \xrightarrow{\text{fast}} x' \tag{92}$$
$$g^P + y' \xrightarrow{\text{fast}} y' \tag{93}$$
$$g^P \xrightarrow{\text{slow}} g^1 + g^2 \tag{94}$$

Copying We use our copier module to implement the line of pseudo-code `z = z + y`.

$$g^1 + y \xrightarrow{\text{slow}} g^1 + y' \tag{95}$$
$$g^1 + y_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{96}$$
$$g^1_{\text{ab}} + y' \xrightarrow{\text{slow}} y + z \tag{97}$$

Decrement We use our decrement module to implement the line of pseudo-code `x = x - 1`.

$$x + g^2 \xrightarrow{\text{slow}} x' + g^2 \tag{98}$$
$$g^2 + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{99}$$
$$2\,x' + g'^2_{\text{ab}} \xrightarrow{\text{fast}} x' + x + x^{\text{rx}} \tag{100}$$
$$x^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{101}$$
$$x' + x^{\text{rx}}_{\text{ab}} + g'^2_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{102}$$

Absence Indicators Four absence indicators are needed by this system; they are of the same form as all others described in this paper.

Appendix: Raise-to-a-Power Reactions

We present chemical reactions that implement the pseudo-code in Figure 1.

System Initialization We assume that an external source injects some quantity of *begin* at the outset. This type is immediately is split into two types, g^1 and g^7, which will be used to copy x to y (for the line of code `y = x`) and to decrement p (for the line of code `p = p - 1`), respectively. This initializations takes care of the steps before the first `while` statement.

$$\textit{begin} \xrightarrow{\text{fast}} g^1 + g^7 \tag{103}$$

Copy x to y $[g^1]$

$$x + g^1 \xrightarrow{\text{slow}} xy + g^1 \tag{104}$$
$$g^1 + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{105}$$
$$xy + g^1_{\text{ab}} \xrightarrow{\text{slow}} x + y \tag{106}$$

Loop Restart Our condition for restarting the main loop is that we still have p present in the system, and that we are not currently somewhere in the middle of the loop. The chemical type *done* is produced at the end of each loop from reactions 165 through 177 below. We also will wait until our post-loop cleanup in reactions 178 through 180 below is complete. At the start of each loop, we produce an injection of g^2 and g^7; these initiate the loop.

$$\begin{aligned} &st_{\text{ab}} + cyc_{\text{ab}} + g^6_{\text{ab}} \\ &+stgo_{\text{ab}} + done + p \xrightarrow{\text{slow}} go^{\text{P}} + done + p \end{aligned} \tag{107}$$
$$go^{\text{P}} + st \xrightarrow{\text{fast}} st \tag{108}$$
$$go^{\text{P}} + cyc \xrightarrow{\text{fast}} cyc \tag{109}$$
$$go^{\text{P}} + g^6 \xrightarrow{\text{fast}} g_6 \tag{110}$$
$$go^{\text{P}} + stgo \xrightarrow{\text{fast}} stgo \tag{111}$$
$$go^{\text{P}} \xrightarrow{\text{slow}} go + stgo \tag{112}$$
$$go \xrightarrow{\text{fast}} g^2 + g^7 \tag{113}$$
$$w + done \xrightarrow{\text{fast}} w \tag{114}$$
$$xw + done \xrightarrow{\text{fast}} xw \tag{115}$$
$$cyc + stgo \xrightarrow{\text{slow}} cyc \tag{116}$$

Copy x to w (once each loop) $[g^2]$ First, we take care of `w = x`.

$$x + g^2 \xrightarrow{\text{slow}} xw + g^2 \tag{117}$$
$$x_{\text{ab}} + g^2 \xrightarrow{\text{slow}} \varnothing \tag{118}$$
$$g^2_{\text{ab}} + xw \xrightarrow{\text{slow}} x + w \tag{119}$$

Loop-Running Indicator We produce a chemical type *cyc* whenever we are executing a loop. This is to ensure that our modules will not inadvertently fire when we do not wish them to do so.

$$w \xrightarrow{\text{slow}} w + cyc \tag{120}$$
$$2\,cyc \xrightarrow{\text{fast}} cyc \tag{121}$$

Multiply Loop Start The inner `while` loop is our multiply operation, handled by the next three groups of reactions.

$$w + w'_{\text{ab}} + yd_{\text{ab}} \xrightarrow{\text{slow}} w + g^{34\text{P}} \tag{122}$$
$$g^{34\text{P}} + w' \xrightarrow{\text{fast}} w' \tag{123}$$
$$g^{34\text{P}} + yd \xrightarrow{\text{fast}} yd \tag{124}$$
$$g^{34\text{P}} \xrightarrow{\text{slow}} g^3 + g^4 \tag{125}$$

Copy y to d (multiply loop) $\left[g^3\right]$

$$y + g^3 \xrightarrow{\text{fast}} yd + g^3 \tag{126}$$
$$g^3 + y_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{127}$$
$$g^3_{\text{ab}} + yd \xrightarrow{\text{slow}} y + d \tag{128}$$

Decrement w $\left[g^4\right]$

$$w + g^4 \xrightarrow{\text{fast}} w' + g^4 \tag{129}$$
$$g^4 + w_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{130}$$
$$g^4_{\text{ab}} \xrightarrow{\text{slow}} g'^4_{\text{ab}} \tag{131}$$
$$2\,w' + g'^4_{\text{ab}} \xrightarrow{\text{fast}} w' + w + w^{\text{rx}} \tag{132}$$
$$w^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{133}$$
$$w' + w^{\text{rx}}_{\text{ab}} + g'^4_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{134}$$
$$2\,g'^4_{\text{ab}} \xrightarrow{\text{slow}} g'^4_{\text{ab}} \tag{135}$$
$$g'^4_{\text{ab}} + g^4 \xrightarrow{\text{fast}} g^4 \tag{136}$$

End of Multiply Detection Once the multiplication operation has completed, we produce g^5, enabling the next step:

$$\begin{aligned} &w_{\text{ab}} + w'_{\text{ab}} + g^2_{\text{ab}} \\ &+xw_{\text{ab}} + st_{\text{ab}} + done_{\text{ab}} \xrightarrow{\text{slow}} g^{5\text{P}} \end{aligned} \tag{137}$$
$$g^{5\text{P}} + w \xrightarrow{\text{fast}} w \tag{138}$$
$$g^{5\text{P}} + w' \xrightarrow{\text{fast}} w' \tag{139}$$
$$g^{5\text{P}} + g^2 \xrightarrow{\text{fast}} g^2 \tag{140}$$
$$g^{5\text{P}} + xw \xrightarrow{\text{fast}} xw \tag{141}$$
$$g^{5\text{P}} + st \xrightarrow{\text{fast}} st \tag{142}$$
$$g^{5\text{P}} + done \xrightarrow{\text{fast}} done \tag{143}$$
$$g^{5\text{P}} + cyc \xrightarrow{\text{slow}} g^5 + cyc \tag{144}$$

Clear y $\left[g^5\right]$ We must take care of the lines `y = d` and `d = 0`. First, we clear our previous quantity of y.

$$g^5 + y \xrightarrow{\text{slow}} g^5 \tag{145}$$
$$y_{\text{ab}} + g^5 \xrightarrow{\text{slow}} \varnothing \tag{146}$$

Inhibit production of g^5 We stop production of g^5 so that we may preserve the quantity of y that we are going to receive from d.

$$y_{\text{ab}} + yd_{\text{ab}} \xrightarrow{\text{slow}} st^{\text{P}} \tag{147}$$
$$st^{\text{P}} + y \xrightarrow{\text{fast}} y \tag{148}$$
$$st^{\text{P}} + yd \xrightarrow{\text{fast}} yd \tag{149}$$
$$st^{\text{P}} + g^5 \xrightarrow{\text{slow}} st + g^5 \tag{150}$$

Set y to d $\left[g^6\right]$ Finally, we transfer d to y, clearing d in the process.

$$y_{\text{ab}} + g^5_{\text{ab}} + yd_{\text{ab}} \xrightarrow{\text{slow}} g^{6\text{P}} \tag{151}$$
$$g^{6\text{P}} + y \xrightarrow{\text{fast}} y \tag{152}$$
$$g^{6\text{P}} + g^5 \xrightarrow{\text{fast}} g^5 \tag{153}$$
$$g^{6\text{P}} + yd \xrightarrow{\text{fast}} yd \tag{154}$$
$$g^{6\text{P}} + d + st \xrightarrow{\text{slow}} g^6 + d + st \tag{155}$$
$$g^6 + d \xrightarrow{\text{slow}} g^6 + y \tag{156}$$

Decrement p $\left[g^7\right]$ The decrement of p is used several in two distinct cases, but we only need one instance of the module for our system.

$$p + g^7 \xrightarrow{\text{fast}} p' + g^7 \tag{157}$$
$$g^7 + p_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{158}$$
$$g^7_{\text{ab}} \xrightarrow{\text{slow}} g'^7_{\text{ab}} \tag{159}$$
$$2\,p' + g'^7_{\text{ab}} \xrightarrow{\text{fast}} p' + p + p^{\text{rx}} \tag{160}$$
$$p^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{161}$$
$$p' + p^{\text{rx}}_{\text{ab}} + g'^7_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{162}$$
$$2\,g'^7_{\text{ab}} \xrightarrow{\text{slow}} g'^7_{\text{ab}} \tag{163}$$
$$g'^7_{\text{ab}} + g^7 \xrightarrow{\text{fast}} g^7 \tag{164}$$

End-of-Loop Detection We know that we have finished a loop when all operations within and prior to the loop have completed.

$$\begin{aligned}
yd_{\mathrm{ab}} + d_{\mathrm{ab}} + go_{\mathrm{ab}} + g^2_{\mathrm{ab}} & \\
+xw_{\mathrm{ab}} + begin_{\mathrm{ab}} + g^1_{\mathrm{ab}} & \\
+xy_{\mathrm{ab}} + g^7_{\mathrm{ab}} + p'_{\mathrm{ab}} &\xrightarrow{\text{slow}} done^{\mathrm{P}} && (165)\\
done^{\mathrm{P}} + yd &\xrightarrow{\text{fast}} yd && (166)\\
done^{\mathrm{P}} + d &\xrightarrow{\text{fast}} d && (167)\\
done^{\mathrm{P}} + go &\xrightarrow{\text{fast}} go && (168)\\
done^{\mathrm{P}} + g^2 &\xrightarrow{\text{fast}} g^2 && (169)\\
done^{\mathrm{P}} + xw &\xrightarrow{\text{fast}} xw && (170)\\
done^{\mathrm{P}} + begin &\xrightarrow{\text{fast}} begin && (171)\\
done^{\mathrm{P}} + g^1 &\xrightarrow{\text{fast}} g^1 && (172)\\
done^{\mathrm{P}} + xy &\xrightarrow{\text{fast}} xy && (173)\\
done^{\mathrm{P}} + g^7 &\xrightarrow{\text{fast}} g^7 && (174)\\
done^{\mathrm{P}} + p' &\xrightarrow{\text{fast}} p' && (175)\\
done^{\mathrm{P}} &\xrightarrow{\text{slow}} done && (176)\\
2\,done &\xrightarrow{\text{fast}} done && (177)
\end{aligned}$$

Post-Loop Cleanup Finally, we reset the system back to its initial state.

$$\begin{aligned}
st + done &\xrightarrow{\text{fast}} done && (178)\\
g^6 + done &\xrightarrow{\text{fast}} done && (179)\\
cyc + done &\xrightarrow{\text{fast}} done && (180)
\end{aligned}$$

Absence Indicators

Twenty-five absence indicators are used by the reactions above. They are generated by the method outlined in the paper and omitted here to save space.

Appendix: Exponentiation Reactions

We present chemical reactions that implement the pseudo-code in Figure 2.

System Initialization As with our multiplication module, we have a small set of reactions to control the overall timing.

$$x + x'_{\text{ab}} + y'_{\text{ab}} \xrightarrow{\text{slow}} x + g^P \tag{181}$$
$$g^P + x' \xrightarrow{\text{fast}} x' \tag{182}$$
$$g^P + y' \xrightarrow{\text{fast}} y' \tag{183}$$
$$g^P \xrightarrow{\text{slow}} g^1 + g^2 \tag{184}$$

Doubling We use a slight variation of our copier module to implement the line of pseudo-code `y = 2 * y`.

$$g^1 + y \xrightarrow{\text{slow}} g^1 + y' \tag{185}$$
$$g^1 + y_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{186}$$
$$g^1_{\text{ab}} + y' \xrightarrow{\text{slow}} 2\,y \tag{187}$$

Decrement As with our multiplication module, we decrement `x` once each loop.

$$x + g^2 \xrightarrow{\text{slow}} x' + g^2 \tag{188}$$
$$g^2 + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{189}$$
$$2\,x' + g'^2_{\text{ab}} \xrightarrow{\text{fast}} x' + x + x^{\text{rx}} \tag{190}$$
$$x^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{191}$$
$$x' + x^{\text{rx}}_{\text{ab}} + g'^2_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{192}$$

Absence Indicators Four absence indicators are needed by this system; they are of the same form as all others described in this paper.

Appendix: Logarithm Reactions

We present chemical reactions that implement the pseudo-code in Figure 3.

System Initialization As with our other modules, we have a small set of reactions to control the overall timing of our system.

$$2\,x + x'_{\text{ab}} + y'_{\text{ab}} \xrightarrow{\text{slow}} 2\,x + g^P \tag{193}$$
$$g^P + x' \xrightarrow{\text{fast}} x' \tag{194}$$
$$g^P + y' \xrightarrow{\text{fast}} y' \tag{195}$$
$$g^P \xrightarrow{\text{slow}} g^1 + g^2 \tag{196}$$

Halving We use a slight variation of our decrement module to implement the operation `x = x / 2`.

$$x + g^1 \xrightarrow{\text{slow}} x' + g^1 \tag{197}$$
$$g^1 + x_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{198}$$
$$2\,x' + g'^1_{\text{ab}} \xrightarrow{\text{fast}} x + x^{\text{rx}} \tag{199}$$
$$x^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{200}$$
$$x' + x^{\text{rx}}_{\text{ab}} + g'^1_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{201}$$
$$\tag{202}$$

Increment We use our increment module to implement the line `y = y + 1`. We set y to be 1 initially.

$$y + g^2 \xrightarrow{\text{slow}} y' + g^2 \tag{203}$$
$$g^2 + y_{\text{ab}} \xrightarrow{\text{slow}} \varnothing \tag{204}$$
$$2\,y' + g'^2_{\text{ab}} \xrightarrow{\text{fast}} y' + y + y^{\text{rx}} \tag{205}$$
$$y^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{206}$$
$$y' + y^{\text{rx}}_{\text{ab}} + g'^2_{\text{ab}} \xrightarrow{\text{slow}} 2\,y \tag{207}$$
$$\tag{208}$$

Cleanup Once the module has completed, we decrement y by one, storing the result in y_f.

$$x'_{\text{ab}} + y'_{\text{ab}} \xrightarrow{\text{slow}} done^{\text{P}} \tag{209}$$
$$done^{\text{P}} + 2\,x \xrightarrow{\text{fast}} 2\,x \tag{210}$$
$$done^{\text{P}} + x' \xrightarrow{\text{fast}} x' \tag{211}$$
$$done^{\text{P}} + y' \xrightarrow{\text{fast}} y' \tag{212}$$
$$done^{\text{P}} \xrightarrow{\text{slow}} done \tag{213}$$
$$2\,done \xrightarrow{\text{fast}} done \tag{214}$$
$$done + 2\,y \xrightarrow{\text{slow}} y + y_f \tag{215}$$

Absence Indicators Two special absence indicators are used by the halving and increment modules above; a total of 13 are needed for the system to function properly. They are of the same form as all other absence indicators, described in the paper.

Appendix: Models and Parameters

All the models described in this paper are contained in an XML file. This file is available at `http://tinyurl.com/rate-indepedent-xml`.

The file is designed for use with Cain, a biochemical simulator from Caltech [14]. It contains initial quantities for all types. All non-zero quantities can be modified as the user desires to simulate different input values. Within Cain, we suggest *Gillespie's Direct Method* for all simulations.

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

Figure 1. Pseudo-code to implement raise-to-a-power operation.

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

Figure 2. Pseudo-code to implement the exponentiation operation.

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

Figure 3. Pseudo-code to implement the logarithm operation.