

# Programmable Catalytic Particle Computers

Anthony M. L. Liekens<sup>1</sup> and Chrisantha T. Fernando<sup>2</sup>

<sup>1</sup> Technische Universiteit Eindhoven, Eindhoven, the Netherlands

<sup>2</sup> Department of Computer Science, University of Birmingham, Edgbaston, UK

**Abstract.** The *Bare Bones* language is a programming language with a minimal set of operations that exhibits universal computation. We present a conceptual framework, Chemical Bare Bones, to construct Bare Bones programs by programming the state transitions of a multi-functional catalytic particle. Molecular counts represent program variables, and are altered by the action of the catalytic particle. Chemical Bare Bones programs are ‘compiled’ and stochastically modeled to undertake computations such as multiplication. The Chemical Bare Bones implementation is naturally suited to parallel computation. The fragility of the program to stochastic noise can be ameliorated by analog control strategies that exist in biochemistry, e.g. bistability of auxiliary particles that help ‘gate’ the state-transitions of the program controller. Analog chemical reaction network components can easily be incorporated to serial programs, resulting in efficient hybrid analog-digital systems.

## 1 Introduction

### 1.1 Chemical computing

An approach for programming a chemical computer is to design a complex ‘particle’ capable of a controlled transition between configurations, where each configuration is capable of catalyzing a specific set of reactions. Ribozymes can be artificially selected that catalyse specific reactions [1]. Multi-enzyme complexes are common in cells, e.g. PDGF and Tar Complexes [2]. Just as we require, they possess multiple catalytic activities and exist in many states. Programability arises because the state of a complex subunit is dependent on the states of other subunits on the complex. The topology of the complex can be designed, e.g. clusters, chains, rings, allowing appropriate ‘conformational spread’ [3]. Approximately digital solid-state circuitry can be produced in proteins [4].

This approach differs from other work in chemical computing with reaction networks in the following ways. Our method, Chemical Bare Bones (CBB) does not make explicit the implementation details of the catalytic reactions. The substrates may be proteins, RNAs or metabolites. Although DNA hybridization catalyst circuits have been proposed by Seelig et al, they model at the algorithm level, circuits of logic gates, not serially executable programs [5]. CBB describes computations carried out by only *one* particle complex with multiple states, and not networks of catalytic particles computing in a distributed manner. Such neural network metaphors utilize coupled cascade cycles, where the

weights are the extent of allosteric and covalent modification of the equilibrium position between binary protein configurations that represent activities [6–8]. Although it is possible to produce logic gates with an enzyme cascade cycle it will be a formidable task to assemble many of these gates together into a network [9]. The demonstration that CBB is Turing universal lies in the isomorphism between chemical reactions and the Bare Bones language. This overlays the underlying Turing universality of chemical kinetics on which our system depends [10]. Other approaches to demonstrating the Turing universality of a chemical computing system depend for example on forming an isomorphism with Wang tiles [11]. CBB does not produce analog reaction networks, e.g. integral feedback controllers [12] or analog networks capable of computing mathematical functions at their steady state [13]. Such Analog networks cannot easily be hand-designed whereas CBB allows hand-design of similar functionalities, plus the incorporation of analog networks where necessary. Evolutionary methods are not required to program CBB.

## 1.2 The Bare Bones programming language

The Bare Bones programming language is a minimal set of instructions that exhibits universal computation [14]. The Bare Bones language only contains 2 assignment statements and one control structure.

The assignment statements are **increase**  $v$  and **decrease**  $v$  where  $v$  denotes a variable name representing a strictly positive integer. Variables are created in memory when they are used for the first time, with a random initial value.

The sole control structure is represented by a specific **while** loop, as a **while**  $v \neq 0$  **do** ... **end while** statement pair. The Bare Bones programming language only knows one condition,  $v \neq 0$ , where  $v$  can be any variable, to control the **while** loop.

Commonly, the syntax for the Bare Bones programming language also includes a **clear**  $v$  statement, which resets the value of variable  $v$  to 0. However, this functionality can also be written with the above statements and control structure as in Algorithm 1.

---

**Algorithm 1** **clear**  $v$  in Bare Bones

---

```
while  $v \neq 0$  do  
  decrease  $v$   
end while
```

---

The Bare Bones programming language can express algorithms for computing all Turing-computable functions, i.e., the Bare Bones programming language is Turing complete.

As an example of a Bare Bones program, Algorithm 2 multiplies the values in variables  $v$  and  $w$  and stores the result in  $u$ . Initially, the program resets the result  $u$  and temporary variables  $t_1$  and  $t_2$  to 0. The program uses these temporaries

to reconstruct the initial values of  $v$  and  $w$  during summation loops. Versions that destroy the initial values of  $v$  or  $w$  during the computation are also possible and result in simpler algorithms.

---

**Algorithm 2** Multiplication ( $u \leftarrow v * w$ ) in Bare Bones

---

```
clear  $u$ , clear  $t_1$ , clear  $t_2$ 
while  $v \neq 0$  do
  increase  $t_1$ , decrease  $v$ 
end while
while  $t_1 \neq 0$  do
  decrease  $t_1$ , increase  $v$ 
  while  $w \neq 0$  do
    increase  $t_2$ , decrease  $w$ 
  end while
  while  $t_2 \neq 0$  do
    increase  $w$ , increase  $u$ , decrease  $t_2$ 
  end while
end while
```

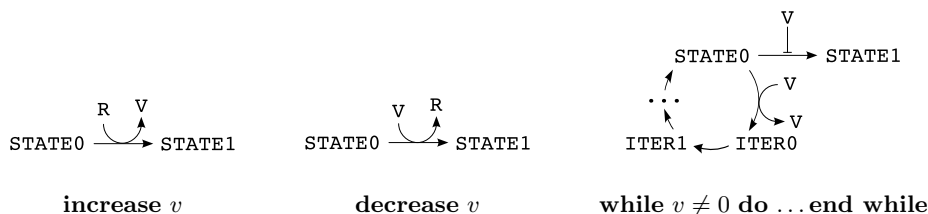
---

## 2 Methods

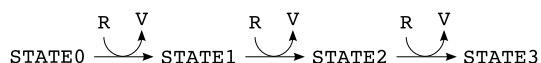
We show a conceptual implementation of Bare Bones' 3 basic operators in networks of chemical reactions. The program operates on molecules and changes their numbers, in similarity to Bare Bones' use of variables and their values. A particular molecule is introduced to control the flow of the program, in similarity to a program counter or instruction counter in computers. The controlling particle reacts with the variables, and moves to a different configuration or conformation representing the next instruction for the program. The basic Bare Bones instructions in reaction networks allow us to construct programs as reaction networks. We construct some basic programs, and show the possibility to use reaction networks as subroutines.

### 2.1 Molecules as variables and flow controllers

Molecules are used for two differing purposes. On one hand, they represent the variables of our program. The number of molecules  $V$  denotes the value of variable  $v$ . We assume that the reaction networks have access to an unlimited amount  $r$  of resource particles  $R$  to maintain mass conservation during the execution of a program in the reaction network. These resource particles  $R$  can be consumed (or produced) to increase (respectively decrease) the amount of molecules representing a variable in the program. Consequently, a reaction that produces a molecule  $V$  out of a resource molecule  $R$  increases the value of  $v$ , and a reaction that reacts a molecule  $V$  to become a resource particle  $R$  decreases  $v$ .



**Fig. 1.** Chemical Bare Bones primitives **increase  $v$** , **decrease  $v$**  and **while  $v \neq 0$  do ... end while**



**Fig. 2.** Chaining three **increase  $v$**  instructions.

Secondly, special molecules act as program counters, and control the flow of the program through the network. A program counter molecule can be in a set of configurations. Each of these configurations corresponds to an instruction in the Bare Bones program. **AML: This would be a great place to add some discussion of what our program controller could be implemented as a complex of proteins or other.**

## 2.2 Bare Bones primitives as reactions

The programming of a network is represented by reactions between molecules in the system. In a reaction, the program counter molecule reacts with variable molecules and produces a different configuration for the program counter molecule denoting the next state of the program. Simple reactions that implement the **increase  $v$**  and **decrease  $v$**  primitives as basic reaction networks are depicted in Figure 1. Both primitives can be written as a single reaction. In the case of **increase  $v$** , the control particle in state **STATE0** reacts with an abundant resource molecule **R**, where a new molecule **V** and the control particle in configuration **STATE1** are the products of the reaction. When the control particle signals **STATE1**, it announces the termination of the **increase  $v$**  program to the system. Similarly, we let the program counter react with particle **V** to release a resource particle **R** to instantiate the **decrease  $v$**  primitive as a reaction network.

These two basic primitives can be chained to increase or decrease multiple times or multiple variables as an ordered series of instructions. As an example, Figure 2 represents a program that increases the value of  $v$  with 3. To construct this program, the controller goes through three consecutive states where it catalyzes the production of one **V** particle. When the controller is in the **STATE3** state, the count of molecules **V** has been augmented by 3 since the **STATE0** state.

The last instruction of the Bare Bones language that remains is the **while  $v \neq 0$  do ... end while** control structure. The loop is implemented as two reactions

as shown in Figure 1. One reaction sets the program counter molecule from state STATE0 to the first instruction, ITER0, of the iteration, if molecules V are present. An iteration of the while loop is a sequence of Bare Bones primitives. At the end of the iteration, the program counter is returned to its STATE0 state. The second reaction that makes up a **while** control structure moves the program counter out of the loop, and is inhibited by V molecules. For now we assume that this inhibition is *strict*, i.e., one V molecule locks the transition of the program counter from its STATE0 state to its STATE1. In a later section, we analyze the behavior of the loop with stochastic, competitive inhibition. Under the assumption of strict inhibition, the instructions in the iteration are executed for as long as there are molecules V in the system, and the loop is exited when no molecules V are present. Note that if the value of  $v$  is not decreased during the iteration, the control structure loops unboundedly.

The above construction allows Bare Bones primitives to be ported to a platform of conceptual catalytic and inhibitory reactions. As a consequence, any Bare Bones program can be implemented as a reaction network. Since the Bare Bones language is Turing complete, our interpretation of the language as chemical reaction networks results in a universal language as well.

Because of this universality, more complex instructions that are being added to the language does not improve its expressive power. Higher level primitives can all be implemented as Bare Bones instructions, but might have simpler interpretations that can be added to the language’s basics, and increase the readability of the programs. As an example, an **if  $v \neq 0$  then ... else ... endif** control structure could be implemented as two consecutive **while** loops, or it can be implemented more straightforward as a pair of reactions. One reaction would move the controller molecule to one series of instructions if V is present, where the second reaction, inhibited by V molecules, moves the pointer to a second sequence. At the end of both sequences of instructions, the controller than points to the instruction that follows our control structure.

### 2.3 Basic programs

**Clear program** Figure 3 shows an implementation of the **clear**  $v$  operator as a basic reaction network. For as long as there are molecules V in the reactor, these react with state STATE0 of the program controller, to produce a resource particle R. If no more molecules V are left in the reactor, the inhibitory reaction becomes unlocked, thereby ending the execution of the program by moving the program controller to STATE1.

**Multiplier** Figure 4 shows an implementation of a multiplier, as in Algorithm 2. The program initially clears the result and temporary variables  $t_1$  and  $t_2$ . Then, molecules  $V$  are moved to temporary molecules T1. Consuming T1, the **while** loop at state STATE4 sums the value of  $v$  to  $u$ ,  $t_1$  times, using a similar loop with temporary molecule T2. When all temporary molecules T1 have been used, the program signals its end by setting the state of the program controller to STATE8.



catalyzed by the state of the program controller, and the program controller has a method to sense the termination of the analog program, then Chemical Bare Bones programs can, in theory, interface with such analog circuits.

### 3 Results

#### 3.1 Reaction networks as machines

**Stochastic models and correctness** To set up general Bare Bones programs in reaction networks, we have previously assumed that the inhibition rule to end a while loop is strict. As a result, programs are executed without error. However, this assumption is not feasible in real reaction networks. Because of real chemistry’s stochastic nature, the correctness of a program is no longer guaranteed.

Assuming a well-stirred mixture, and a basic model of mass-action kinetic laws, a reaction is said to occur with a propensity proportional to its reaction rate and the number of reactants available in the system. If we assign a sufficiently fast reaction rate to the reaction that starts an iteration of the while loop, and a relatively slow reaction rate to the competitively inhibitory reaction that exits the while loop, we can decrease the probability that a while loop is exited prematurely.

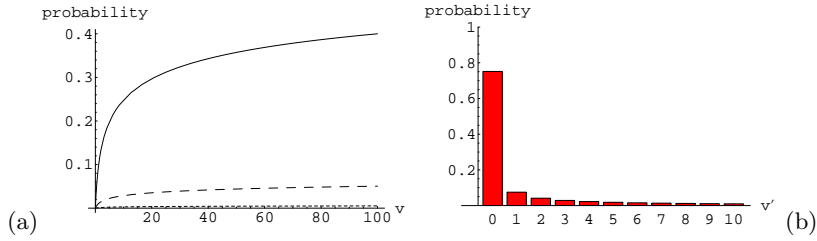
For now we suppose that the program counter molecule of the program points to the start of the **while**  $v \neq 0$  **do** . . . **end while** loop, where the program can either enter an iteration of the **while** loop, or exit the loop. We want to determine the probability that the program exits the **while** loop for a given number of  $V$  particles. Let  $k_{\text{fast}}$  be the reaction rate for the reaction that enters an iteration of the loop. The propensity for this reaction to occur is  $k_{\text{fast}}v$ . Similarly, let  $k_{\text{slow}}$  be the reaction rate to exit the **while** loop, and transform the program counter molecule to its next state, ending the **while** loop. Assuming competitive inhibition, the propensity of this reaction is  $k_{\text{slow}}$ . Using Gillespie’s algorithm [15] for stochastic models of reactions, we can determine the probability that the system exits the **while** loop prematurely, dependent on the number of molecules  $V$  in the reactor. The probability that either reaction occurs is proportional to its propensity, i.e.

$$\Pr[\text{next reaction enters iteration}] = \frac{k_{\text{fast}}v}{k_{\text{fast}}v + k_{\text{slow}}} \quad (1)$$

$$\Pr[\text{next reaction exits loop}] = \frac{k_{\text{slow}}}{k_{\text{fast}}v + k_{\text{slow}}}. \quad (2)$$

If  $v = 0$ , the probability that the next reaction exits the **while** loop is 1. When the number of molecules  $V$  is low, the probability to prematurely end the **while** loop is highest. The probability that the **while** loop stops iterating is lower if a faster reaction rate  $k_{\text{fast}}$  or slower rate  $k_{\text{slow}}$  is chosen.

We can, as an example, compute the probability that a **clear**  $v$  program, as in Figure 3, stops removing  $V$  elements and moves to its end state(STATE1) too



**Fig. 5.** (a) Probability of prematurely exiting a **clear**  $v$  program, dependent on the initial value of  $v$ . The continuous, dashed and dotted graphs represent the probabilities for  $k_{\text{fast}}/k_{\text{slow}} = 10, 100$  and  $1000$ , respectively. (b) Probability distribution over the possible end results  $v'$  of a **clear**  $v$  program with initial  $v = 10$ , with ill-defined parameters  $k_{\text{fast}}/k_{\text{slow}} = 10$

early. The probability that the loop is terminated prematurely, is given by

$$1 - \prod_{n=1}^v \frac{k_{\text{fast}} n}{k_{\text{fast}} n + k_{\text{slow}}} = 1 - \frac{v! \Gamma\left(\frac{k_{\text{slow}}}{k_{\text{fast}}} + 1\right)}{\Gamma\left(\frac{k_{\text{slow}}}{k_{\text{fast}}} + v + 1\right)}. \quad (3)$$

Figure 5(a) depicts the probability of terminating a **clear**  $v$  prematurely. For higher  $k_{\text{fast}}/k_{\text{slow}}$  ratios, the probability of incorrectly exiting the while loop is lower. As more iterations of the **while** loop have to be iterated, the probability of exiting too early is higher. Independent on the kinetic rate settings, as  $v$  goes to infinity, the probability of exiting the loop prematurely goes to 1.

The probability to end the **clear**  $v$  program when there are still  $v'$  particles left, with  $1 \leq v' \leq v$  is given by

$$\frac{k_{\text{slow}}}{k_{\text{fast}} v' + k_{\text{slow}}} \prod_{n=v'+1}^v \frac{k_{\text{fast}} n}{k_{\text{fast}} n + k_{\text{slow}}} = \frac{k_{\text{slow}} \Gamma\left(\frac{k_{\text{slow}}}{k_{\text{fast}}} + v'\right) \Gamma(v+1)}{\Gamma\left(\frac{k_{\text{slow}}}{k_{\text{fast}}} + v + 1\right) \Gamma(v'+1)} \quad (4)$$

Figure 5(b) shows the probability to end a **clear**  $v$  operation with  $v'$  particles left, where the initial value of  $v$  is set to 10. Parameter  $k_{\text{fast}}$  was chosen to be 10 times bigger than  $k_{\text{slow}}$ . Increasing the rate between these parameters results in higher probability to terminate the computation correctly, as shown in Figure 5(a).

AML: There should be something here on bistable auxiliary particles to increase the fragility, but I'm not sure how to write that down, as it was one of your ideas :)

**Time complexity** As a result of the previous section, a larger ratio between kinetic rates  $k_{\text{fast}}/k_{\text{slow}}$  increases the accuracy of programs that use **while** loops. There is however a trade-off in the expected running time of the program. Indeed,

for high  $k_{\text{fast}}/k_{\text{slow}}$  ratios, entering iterations of a **while** loop is fast, where exiting from a while loop is relatively very slow.

As an example, we analyze the running time of a successful **clear**  $v$  operation. Assuming that  $k_{\text{fast}}$  and  $k_{\text{slow}}$  are well-separated, the expected running time of the **clear**  $v$  program can be approximated with the sum of expected times that separate consecutive operations, with

$$E[\text{running time of successful } \mathbf{clear} \ v] \approx \frac{1}{k_{\text{slow}}} + \sum_{n=1}^v \frac{1}{nk_{\text{fast}}}. \quad (5)$$

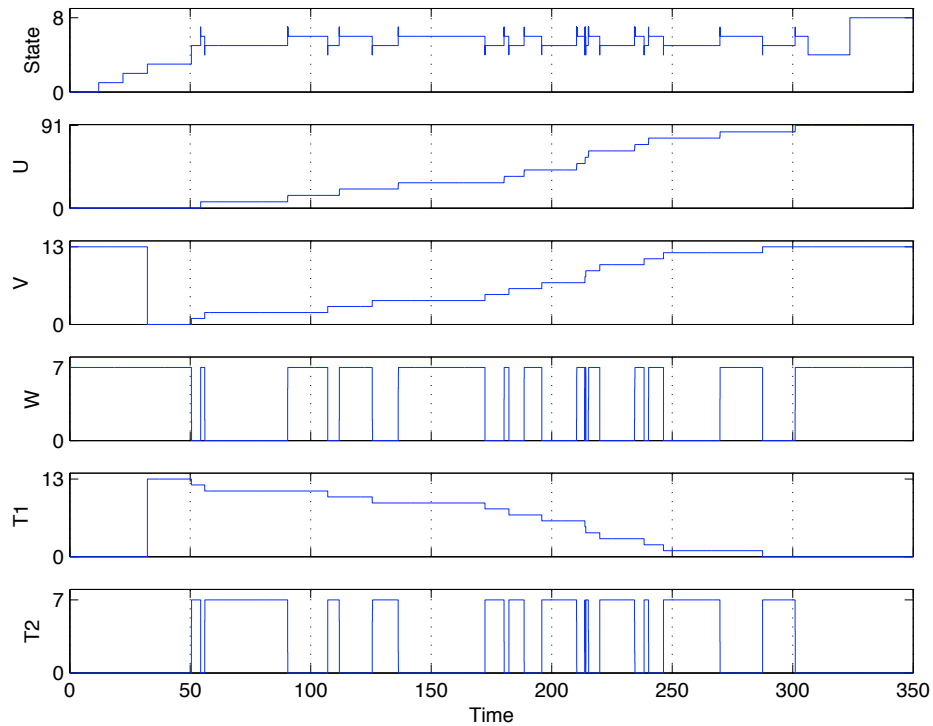
The first term denotes the expected time to exit the while loop when there's no  $V$  particles left, the second term sums up the expected times to execute an iteration of the **while** loop. Note that this is an approximation since we do not take the probability to exit the **while** loop prematurely into account. Since rate  $k_{\text{slow}}$  is chosen to be very small in comparison with  $k_{\text{fast}}$ , the expected running time of the **clear**  $v$  operator is dominated by the time it takes to exit the **while** loop, and not so much by carrying out the instructions in the **while** loop.

This result implies a unique property of Chemical Bare Bones programs with respect to time complexity. By choosing high  $k_{\text{fast}}/k_{\text{slow}}$  ratios, the time complexity of Bare Bones programs in stochastic models becomes dependent on the number of **while** loops that need to be exited during a run, and not on the number of instructions that have to be processed by the program, since these are relatively fast in comparison with the time needed to exit a **while** loop.

**Parallelism** In the above sections, we have constructed essentially serial programs for an inherently parallel platform. By introducing multiple program counter molecules in the reactor, parallel programs can be carried out in reaction networks. However, these multiple program counters and their instructions act on shared memory variables. As a result, the inattentive parallelization of a program may result in the incorrect implementation of an algorithm.

Multi-threaded programs with shared memory require a mutual exclusion concept to control the flow of the program. The simplest implementation is to use semaphores as safeguards of critical sections, i.e., sections of the program that access shared memory. Before a thread of the parallel program executes a critical section, it reserves a semaphore which offers mutual exclusive access to the shared memory. At the end of the critical section, the semaphore is released to allow other threads to access shared memory.

The Bare Bones language in reaction networks allows for the implementation of parallel programs with semaphores. Threads of the parallel program can be started by reactions that produce additional program counter molecules and they can be terminated by consuming these program counters. For the implementation of a semaphore, a variable  $s$  is initialized with value 1 at the beginning of the program. If a thread wants to enter a critical section, it can reserve the semaphore with a **decrease**  $s$  statement. At the end of its critical section, the thread must release the semaphore by an **increase**  $s$  statement to allow other threads to

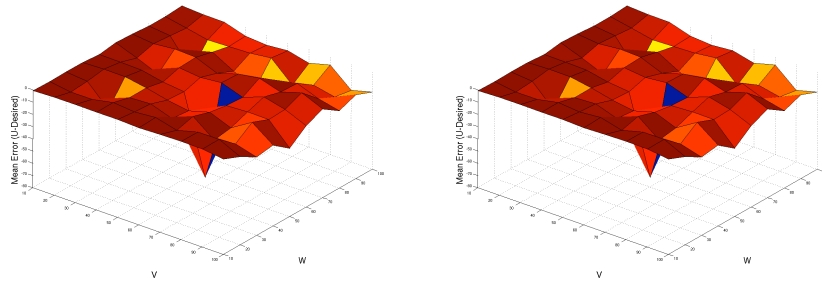


**Fig. 6.** Typical behaviour of the multiplier with initial values  $v = 13, w = 7, u = t_1 = t_2 = 0$ . The end result,  $u = vw = 91$  is achieved when the program controller reaches state 8. The intra-loop rates are fast, with reaction rates  $k_{\text{fast}} = 100$ , whilst the between loop state transition steps are slow, with reaction rate  $k_{\text{slow}} = 0.1$ .

enter their critical sections after locking the semaphore. If the semaphore was reserved by a thread, other threads that request the semaphore with a **decrease**  $s$  reaction are implicitly put on hold since the reaction can only be completed if the semaphore is available.

### 3.2 Simulation

Using the stochastic simulator BioNetS [16] we modeled the multiplier network in Figure 4. The code for running these simulations is available at [www](http://www). Figure 6. shows the behaviour of the multiplier. Figure 7 shows the approximate mean errors of the same network for values of  $V$  and  $W$  in the range 1 to 100. For each  $(V, W)$  pair, 30 trials were conducted with different random seeds.



**Fig. 7.** The vertical axis shows the mean( $U_{end} - (v.w)$ ) obtained over 30 trials for each  $v, w$  pair. The errors are always to underestimate the value of  $v.w$ , due to a **while** loop being exited prematurely. **AML:** Could we have two graphs next to each other for e.g.  $k1/k2=100$  and  $k1/k2=1000$ ?

## 4 Discussion

**Todo:** a general conclusion: Construction allows for the compilation of Bare Bones to networks of reactions

**Todo:** trade-off between speed and accuracy

**Todo:** What about evolvability?

**Acknowledgements** We would like to thank Huub ten Eikelder for helpful discussion regarding the correctness of programs, and semaphores in parallel settings. This work is supported by the European Community through the Evolving Cell Signalling Networks in Silico project (ESIGNET) of the Sixth Framework Programme.

## References

1. Bartel, D., Szostak, J.: Isolation of new ribozymes from a large pool of random sequences. *Science* **261** (1991) 1411–1418
2. Bray, D.: Signaling complexes: Biophysical constraints on intracellular communication. *Annu. Rev. Biophys. Biomol. Struct.* **27** (1998) 59–75
3. Bray, D., Duke, T.: Conformational spread: The propagation of allosteric states in large multiprotein complexes. *Annu. Rev. Biophys. Biomol. Struct.* **33** (2004) 53–73
4. Graham, I., Duke, T.: The logical repertoire of ligand-binding proteins. *Phys. Biol* **2** (2005) 159–165
5. Seelig, G., Yurke, B., Winfree, E.: Dna hybridization catalysts and catalyst circuits. *DNA* (2004) 329–343
6. Bray, D.: Protein molecules as computational elements in living cells. *Nature* **376** (2004) 307–312

7. Arkin, A., Ross, J.: Computational functions in biochemical reaction networks. *Biophysical Journal*. **67** (1994) 560–578
8. Hjelmfelt, A., Weinburger, E., Ross, J.: Chemical implementation of neural networks and turing machines. *Proc. Natl. Acad. Sci. USA*. **88** (1991) 10983–10987
9. Baron, R., Lioubashevski, O., Katz, E., Niazov, T., Willner, I.: Elementary arithmetic operations by enzymes: A paradigm for metabolic pathway-based computing. *Angew. Chem. Int. Ed.* (in press) (2006)
10. Magnasco, M.: Chemical kinetics is turing universal. *Phys. Rev. Lett.* **68** (1997) 1190–1193
11. Winfree, E.: Dna computing by self-assembly. URL: <http://www.nae.edu/nae/bridgecom.nsf/weblinks/MKUF-5UZJFP?OpenDocument> (2003)
12. Sauro, H., Kholodenko, B.: Quantitative analysis of signaling networks. *Prog Biophys Mol Biol*. **86** (2004) 5–43
13. Deckard, A., Sauro, H.: Preliminary studies on the in silico evolution of biochemical networks. *Chembiochem*. **5** (2004) 1423–1431
14. Brookshear, J.G.: *Theory of Computation, Formal Languages, Automata and Complexity*. Benjamin-Cummings (1989)
15. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. *J. Chem. Phys.* **81** (1977) 2340–2361
16. Adalsteinsson, D., McMillen, D., Elston, T.: Biochemical network stochastic simulator (bionets): software for stochastic modeling of biochemical networks. *BMC Bioinformatics* **5** (2004) 24