A Compositional Approach to the Stochastic Dynamics of Gene Networks

Ralf Blossey[1], Luca Cardelli[2], and Andrew Phillips[2]

[1] Interdisciplinary Research Institute, Villeneuve d'Ascq, France
[2] Microsoft Research, Cambridge, United Kingdom

Abstract. We propose a compositional approach to the dynamics of gene regulatory networks based on the stochastic π-calculus, and develop a representation of gene network elements which can be used to build complex circuits in a transparent and efficient way. To demonstrate the power of the approach we apply it to several artificial networks, such as the repressilator and combinatorial gene circuits first studied in Combinatorial Synthesis of Genetic Networks [1]. For two examples of the latter systems, we point out how the topology of the circuits and the interplay of the stochastic gate interactions influence the circuit behavior. Our approach may be useful for the testing of biological mechanisms proposed to explain the experimentally observed circuit dynamics.

1 Introduction

Within the last years a general consensus has emerged that noise and stochasticity are essential building elements of gene regulatory networks. A quantitative understanding of their role is thus needed to understand gene regulation. Regulatory functions can indeed work to eliminate stochastic effects [2], or to even exploit them [3].

In line with new experimental techniques to measure and quantify such behavior, efficient ways to model and simulate gene networks need to be developed, which are currently lacking. Simulations based on differential equations for the concentrations of the various biomolecules, the long-time standard of modeling in biochemical systems, are not well suited for this purpose, except in particular cases. Stochastic effects, which are typically important when molecule numbers are small, are difficult to build into such approaches, and the resulting stochastic equations are time-consuming to simulate. In addition, differential equation models are inherently difficult to change, extend and upgrade, as changes of network topology may require substantial changes in most of the basic equations.

In this paper, we follow a different route. It has recently emerged within computer science in the context of process calculi, and their applications to biological systems. Process calculi [4] are essentially programming languages designed to describe concurrent, interactive systems such as mobile communication networks. Among the various process calculi, π-calculus is one of the best studied because of its compactness, generality, and flexibility. Stochastic variants have appeared recently that address biochemical modeling [5]; they have been used to model molecular interactions [6][7], compartments [8][9], and metabolism [10]. A remaining challenge

C. Priami et al. (Eds.): Trans. on Comput. Syst. Biol. IV, LNBI 3939, pp. 99 – 122, 2006.
© Springer-Verlag Berlin Heidelberg 2006

is to model gene networks, to fully demonstrate the flexibility of process calculi, and to eventually support the integration of molecular, gene, and membrane networks in a single framework.

Here, we introduce process calculi by example, in the context of gene networks; technical details of the approach can be found in the Appendix. Modeling with process calculi is very much like programming. It is carried out in concurrent, stochastic programming languages that can easily support very complex and detailed models in a modular ("compositional") way, where separate program units correspond to separate biochemical components.

Our purpose here is in part tutorial: we aim to show that we can do things *simply* to start with, and already get interesting insights. Models in which molecular details are explicitly treated can be built when needed; e.g. see [11] for a discussion of transcription-translation in phage lambda. In addition to our approach being on the level of gene gates rather than molecular components, we have chosen a style of presentation which we believe will be helpful to researchers from neighbouring disciplines (physics, mathematics and theoretical biology), for whom the existing literature on the application of the stochastic π-calculus may be too demanding.

The paper is structured as follows. We first explain how to represent gene network elements as processes in the stochastic π-calculus and how to execute them. We then apply this representation to model gene networks of increasing complexity, and study some of their behavior. In particular, we address the repressilator circuit [12] and two of the (still controversial) examples of combinatorial circuits first discussed in [1].

2 Modeling Gene Network Elements

2.1 Nullary Gates

We begin by modeling genes that have constitutive transcription but no regulatory control. We focus on the *actions* that are involved in the functioning of genes and molecular components. The generic term *process* is used for any mechanism performing actions and thus progressing through distinct states.

A nullary-input gate (Figure 1), given by a process written *null*(*b*), has a single parameter *b* that represents its transcription product; it takes no input from the

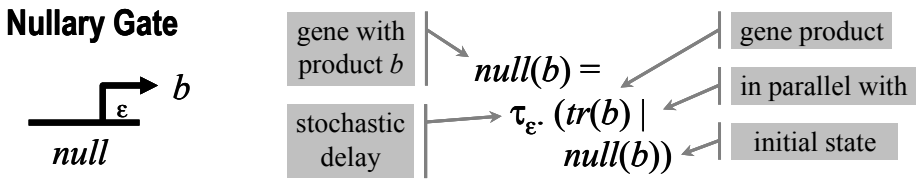


Fig. 1. A gene, *null*(*b*) with constitutive transcription, but no regulation (nullary). The product is a translated protein, *tr*(*b*) that attaches to a binding site *b* on some other gene; the definition of *tr*(*b*) is given later. The definition of *null*(*b*) says that this gate waits for a stochastic *delay* ('τ') of rate ε, and *then* ('.') evolves into two processes in *parallel* ('|'); one is *tr*(*b*), and the other again *null*(*b*), the initial state.

environment. The initial action performed by such a gate is a stochastic delay, τ_ε, where τ is a symbol indicating delay and ε is the stochastic reaction constant, which gives the probability per unit time that the delay action will occur [14]. In general, each action in the π-calculus is associated with a corresponding stochastic reaction rate, such that when an action with rate r is enabled, the probability that it will happen within a period of time t is $F(t) = 1\text{-}e^{-rt}$ [15]. This distribution exhibits the memoryless property, as is required for the Markov property of the stochastic dynamics.

After such a delay action, the original process *null*(*b*) becomes (i.e., changes state to) two separate processes in parallel (separated by the operator "|"): *tr*(*b*) and *null*(*b*). The second process is a copy of the original process *null*(*b*), which was consumed when performing its initial action. The first process, *tr*(*b*), described shortly, represents a molecule of a transcription factor for a binding site *b* on some gene. All together, the *null*(*b*) process is defined as $\tau_\varepsilon.\ (tr(b) \mid null(b))$. A stochastic simulation of a *null*(*b*) process on its own produces multiple copies of *tr*(*b*) at stochastic time intervals characterized by ε, with exactly one copy of *null*(*b*) being preserved.

2.2 Gene Products

We now describe the transcription factor *tr*(*b*) (Figure 2), introducing the process calculus notions of interaction and stochastic choice. Except for delays τ, which happen autonomously, any action that a process performs must happen in conjunction with a complementary action performed by another process. The simultaneous occurrence of complementary actions is an *interaction*, e.g. between two molecules, or between a transcription factor and a promoter site. An action can be *offered* at any time, but only complementarily offered actions can result in actual interactions. For an interaction site, or *channel*, *b*, such complementary actions are conventionally called *input on b* (written '?*b*'), and *output on b* (written '!*b*'). (In our examples we need only consider such simple *signaling* interactions; in general an interaction can also exchange data in the form of a message from output to input.) Hence, ?*b* and !*b* are complementary actions that can exchange a signal between them and allow two corresponding processes to change state.

The transcription factor *tr*(*b*) offers a choice of two actions; one is an output action !*b*, representing interaction with a binding site, and the other is a delay τ, followed by

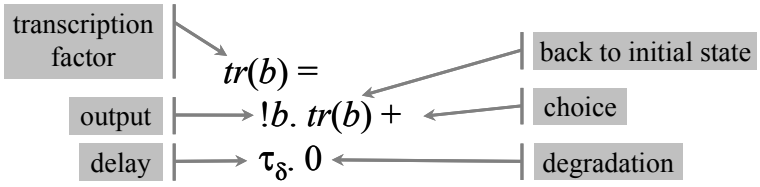


Fig. 2. A transcription factor *tr*(*b*) makes a *stochastic choice* ('+') between either binding to an available promoter site *b* by an *output action* ('!*b*'), or *delaying* ('τ') with rate δ. In the first case, the output action interacts with a corresponding input action at a promoter site *b*, and *then* ('.') the transcription factor returns to its initial state *tr*(*b*), ready to interact again. In the second case, the transcription factor degrades to the inert process ('0').

degradation. These two actions are in a stochastic race, indicated by '+': b has (implicitly defined with it) a fixed associated rate r, and τ has a specific rate δ. If $!b$ wins the race, it means that an interaction has occurred with an input action $?b$ offered elsewhere, and the process returns to the initial state, $tr(b)$. If τ wins the race, however, the following state is 0: the inert process that never performs any actions.

All together, $tr(b)$ is defined as $(!b.\ tr(b)) + (\tau_\delta.0)$, which means that $tr(b)$ has the potential to interact multiple times with promoter sites, but each time (and particularly if no promoter site is available) it has a chance to degrade. Without interactions with binding sites, a fixed population of transcription factors will simply exponentially degrade. If the population is being replenished, then a stable level may be found between production and degradation.

2.3 Unary Gates

We now consider gates with simple regulation. A $neg(a,b)$ gate has a promoter site a with negative regulation (inhibition), and a product b.

The $neg(a,b)$ gate (Figure 3) has a subprocess that is essentially identical to the $null(b)$ gate, i.e., it provides constitutive transcription. However this subprocess is now in a stochastic race with a subprocess $?a.\ \tau_\eta.\ neg(a,b)$. That is, it is in a race with a promoter binding, $?a$. If the promoter component wins the race (by interacting with a transcription factor $tr(a)$), the + choice is taken on the promoter side, and the whole process becomes $\tau_\eta.\ neg(a,b)$. In this state, the gate is stuck performing a stochastic delay τ_η, i.e., it is inhibited, after which it goes back to be $neg(a,b)$.

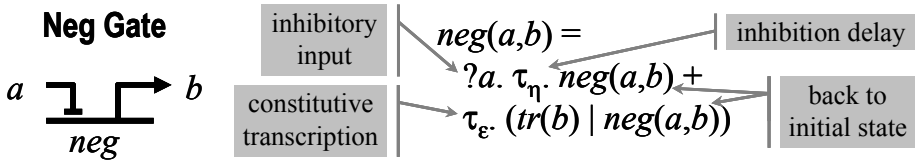


Fig. 3. A gene gate with inhibitory control, $neg(a,b)$ makes a stochastic choice ('+') between constitutive transcription and inhibitory stimulation. The constitutive transcription case (bottom line) is exactly as in Figure 1, but this time it is in a race with a stimulus. If an interaction happens with the input action '$?a$', then the gate enters a stochastic delay ('τ_η'), during which the gate is inhibited, and then returns to the initial state.

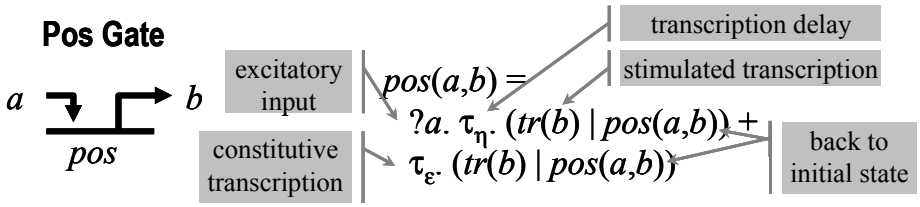


Fig. 4. A gene gate with excitatory control, $pos(a,b)$. This is almost identical to $neg(a,b)$, but the input stimulus '$?a$' is followed by the production of $tr(b)$ instead of an inhibitory delay.

The *pos*(*a*,*b*) gate (Figure 4) has a promoter site *a* with positive regulation (stimulation), and a product *b*. It is similar to the *neg* gate, but instead of an inhibition delay, we have a transcription delay followed by stimulated production of *tr*(*b*).

3 The Stochastic π-Calculus Execution Model

3.1 Simulation Language

We have seen how a biological system can be modeled in the stochastic π-calculus, by representing each component of the system as a process *P* that precisely describes what the component can do. To summarize, the most basic process form is a choice $\Sigma = P_1 + \ldots + P_n$ between zero or more outputs $!x(n)$, inputs $?x(m)$, and delays τ that the component can perform (in the general form of input/output, *n* is the output message and *m* is the input variable). Two components *P* and *Q* can be combined together using parallel composition *P*|*Q*. Channels can be established to allow the components to interact by complementary inputs and outputs. Once a biological system has been modeled using these basic components, the model can be stochastically simulated in order to predict the evolution of the system over time. In this paper, the simulations were obtained using the Stochastic Pi Machine (SPiM), which is described in [13].

Another basic operator of stochastic π-calculus, which we do not need to discuss in detail in this paper, allows the creation of fresh channels. The operator $new\ x_\varepsilon.\ P$ creates a fresh channel *x* of rate ε to be used in the process *P*. The rules of stochastic π-calculus ensure that a "fresh" channel so obtained does not conflict with any other channel. We mention the channel creation operator here just because it allows us to obtain the stochastic delay τ_ε as a derived operator. In fact, we can define:

$$\tau_\varepsilon.P + Q \quad = \quad new\ x_\varepsilon.\ (!x.0 \mid (?x.P + Q)) \qquad \text{for } x \text{ not occurring in } P \text{ or } Q$$

That is, a delay is equivalent to a single communication on a fresh channel of the same rate. Hence, stochastic delays can be reduced to ordinary channel communication, and can be handled uniformly like any other communication, e.g., for simulation purposes.

3.2 Simulator

The Stochastic Pi Machine simulates a given process *P* by first converting the process to a corresponding simulator data structure, consisting of a list of components $A = \Sigma_1, \ldots, \Sigma_M$. The resulting list is then processed by the simulator, by first using a function *Gillespie*(*A*) to stochastically determine the next interaction channel *x* and the corresponding reaction time τ. Once an interaction channel *x* has been chosen, the simulator uses a *selection operator* to randomly select from the list *A* a component of the form $\Sigma + ?x(m).P$ containing an input on channel *x*, and different component of the form $\Sigma' + !x(n).Q$ containing an output on *x*. The selected components can then interact by synchronizing on channel *x*, and the processes *P* (with the input variable *m* replaced by *n*) and *Q* are added to the remainder of the list. The simulator continues processing the list in this way until no more interactions are possible.

The function *Gillespie*(*A*) is based on [14], which uses a notion of *channel activity* to stochastically choose a reaction channel from a set of possible channels. The activity of a reaction channel corresponds to the number of possible combinations of reactants on the channel; channels with a high activity and a fast reaction rate have a higher probability of being selected. A similar notion of activity is defined for the Stochastic Pi Machine, where $Act_x(A)$ denotes the number of possible combinations of inputs and outputs on interaction channel x in a list of components A:

$$Act_x(A)=(In_x(A)*Out_x(A))-Mix_x(A)$$

$In_x(A)$ and $Out_x(A)$ are defined as the number of available inputs and outputs on interaction channel x in A, respectively, and $Mix_x(A)$ is the sum of $In_x(\Sigma_i)\times Out_x(\Sigma_i)$ for each component Σ_i in A. The formula takes into account the fact that an input and an output in the same component cannot interact, by subtracting $Mix_x(A)$ from the product of the number of inputs and outputs on x.

The Stochastic Pi Machine has been formally specified in [13], and the specification has been proved to correctly simulate π-calculus processes. The simulator has also been used to simulate a wide variety of chemical and biological systems. In particular, many of the benchmark examples that were used to validate the Gillespie algorithm [14] have been modeled as π-calculus processes and correctly simulated in SPiM.

3.3 Interaction-Oriented Simulation vs. Reaction-Oriented Simulation

The Gillespie algorithm was originally used to simulate a set of chemical reaction equations expressed in terms of reactants and products, and the results of a simulation were plotted as the quantity of each chemical species versus time. In contrast, the π-calculus does not describe an equation for each type of chemical reaction, but instead describes the behavior of each component in terms of the inputs and outputs it can perform on a set of interaction channels. This gives rise to an interaction-oriented model, as opposed to a chemical-reaction-oriented model, in which a reactant is defined as an input or output on a given interaction channel. Once the notion of a reactant has been defined in this way, the Gillespie algorithm can be directly applied to a given π-calculus model of the biological system. The corresponding simulation results can be plotted as the quantity of each reactant versus time.

4 Gene Networks

4.1 Simple Circuits

In Section 2 we have described gene gates with one input; gates with n inputs can be defined similarly, to form a larger library of components. Once the components are defined, gene circuits can be assembled by providing interaction channels, with associated interaction rates, connecting the various gates. If we write, e.g., *pos*(*a*,*b*) | *neg*(*b*,*c*), the *pos* process will offer output actions !*b*, through *tr*(*b*), and the *neg* process will offer input actions ?*b*. Hence the shared channel *b*, given to both *pos* and *neg* as a parameter, can result in repeated interactions between the two processes over *b*, and hence in network connectivity.

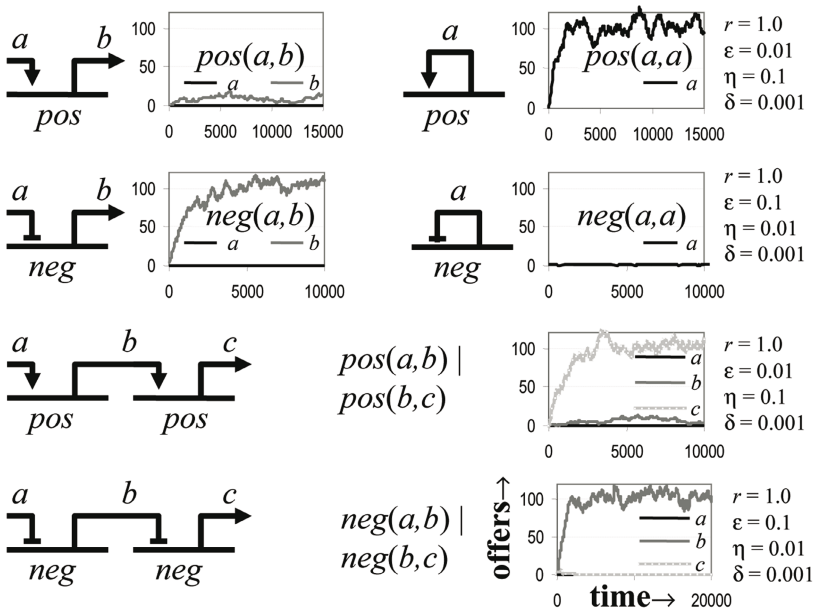


Fig. 5. Compositions of gates represent circuits (left) that exhibit behaviors (right). The channels *a,b,c* are declared separately (not shown) along with their associated stochastic interaction rates. In all simulations, the common rate *r* for *a,b,c* is set to a baseline value of 1.0. The other chosen rates are as indicated in the individual simulations; the fact that they are chosen at simple order-of-magnitude intervals suggests that they are not critical for the intended behavior. The vertical axis is the number of outstanding *offers* of communication: for a channel *a* we may plot output offers !*a* or input offers ?*a*. In all cases above, the networks get started by constitutive transcription only. All the plots are of individual simulator runs.

The simplest circuits we can build are single gates interacting with themselves in a feedback loop, like *pos*(*a*,*a*) (Figure 5). In absence of any stimulus on *a*, *pos*(*a*,*a*), must choose the constitutive transcription route and evolve into *tr*(*a*) | *pos*(*a*,*a*), where now *tr*(*a*) can stimulate *pos*(*a*,*a*) at a faster rate than the constitutive rate, and possibly multiple times. Depending on the production and degradation rates, a stable high level of *tr*(*a*) may be reached. Similarly *neg*(*a*,*a*) can stabilize at a low quantity of *tr*(*a*) where degradation of *tr*(*a*) balances inhibition. A convenient high-signal level of about 100 is maintained in our examples by appropriate rates (see parameters in Figure 5).

The combination *pos*(*b*,*a*) | *neg*(*a*,*b*) (Figure 6) is a self-inhibition circuit, like *neg*(*a*,*a*), and it similarly has a stable output. But now there are two separate products, *tr*(*a*) and *tr*(*b*), so the system (again in absence of any stimulus) can stochastically start with a prevalence of *tr*(*a*) or a prevalence of *tr*(*b*): this can be seen at the beginning of the two plots, before stabilization.

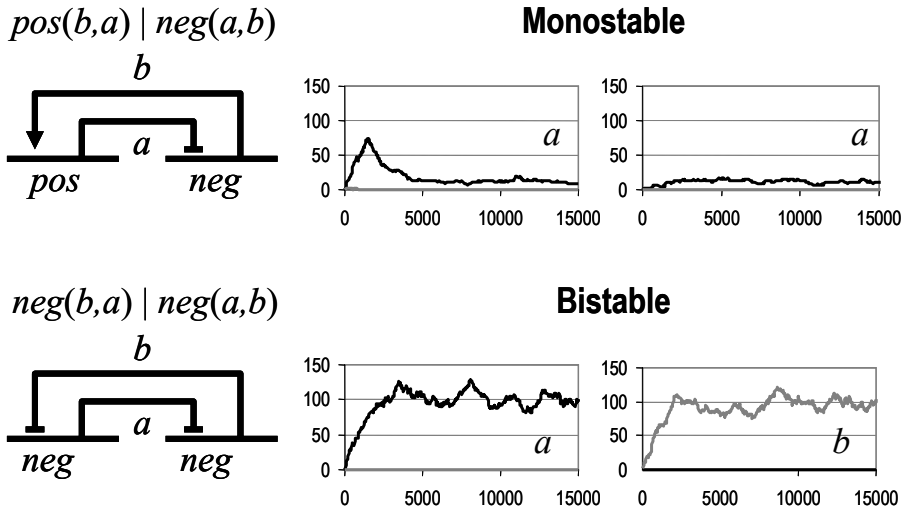


Fig. 6. Feedback loops that are *monostable* (resulting in a single stable state with *a* high after a transient) and *bistable* (resulting in two distinct stable states with *a* high or *b* high)

The combination *neg*(*b*,*a*) | *neg*(*a*,*b*) (Figure 6) is a bistable circuit, which can start up in one state or another, and (usually) stay there.

4.2 Repressilator

The well-known repressilator circuit [12], consisting of three *neg* gates in a loop, is an oscillator. We compare here three different degradation models, aiming to justify somewhat our initial definition for *tr*(-). In the first model (Figure 7(A)), each transcription factor interacts exactly once, and only then it disappears. The repressilator circuit oscillates nicely but, without stochastic degradation, the plots appear very "mechanical"; moreover, the quantities of products grow at each cycle because products do not disappear unless they interact. In the second model (Figure 7(B)), each transcription factor interacts exactly once, or can degrade. Again the plots look mechanical, but the stochastic degradation defines a stable level of product. The third model (Figure 7(C)), with multiple interactions and stochastic degradation, is more realistic and gives more convincing plots. See the Appendix for the simulator script.

The progressive refinement of the definition of *tr*(-), provides an illustration of how one can play with process descriptions to find models that show a balance between simplicity and realism. A further step could be to model both attachment and detachment of transcription factors, and then to model both transcription and translation.

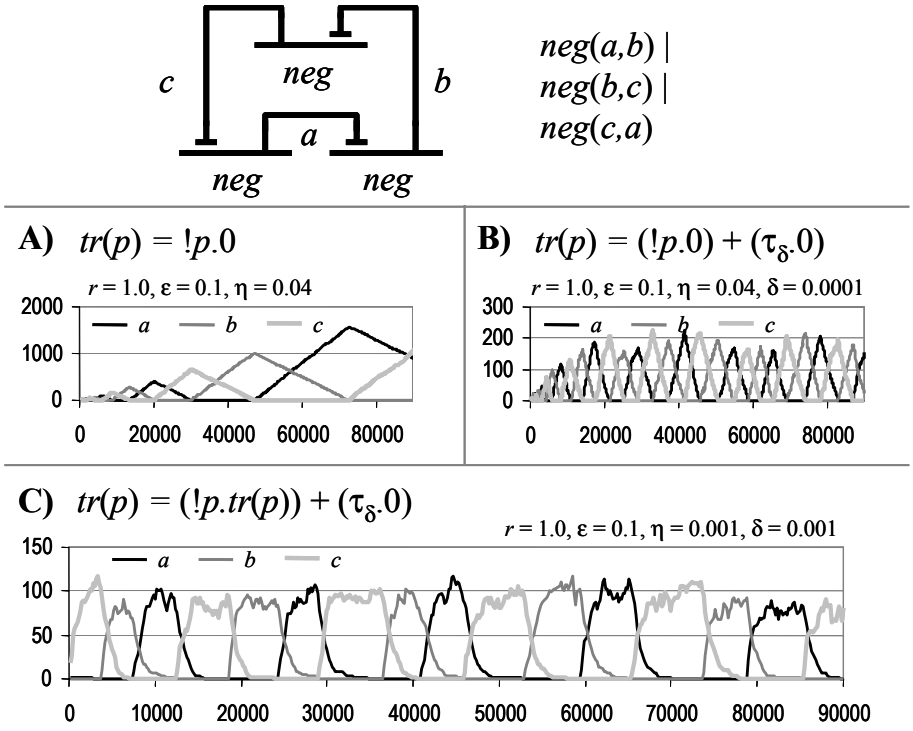


Fig. 7. The Repressilator circuit and its dynamics for different degradation models (A – C). The detailed explanation is found in the text.

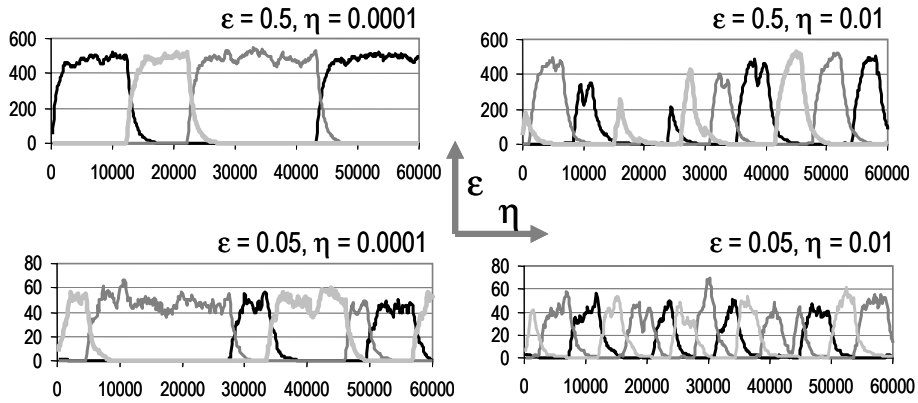


Fig. 8. Repressilator frequency and amplitude, regulated by η and ε. *Cf.* Figure 7(C).

4.3 Network Properties: Oscillation

It is instructive to take a "systems" approach and see what the rate parameters described earlier mean in the context of networks of gates. In the case of the

repressilator we can see that the constitutive rate (together with the degradation rate) determines oscillation amplitude, while the inhibition rate determines oscillation frequency. Figure 8 shows the variation of ε and η from their values in Figure 7(C)); note the differences in scale.

Moreover, we can view the interaction rate *r* as a measure of the volume (or temperature) of the solution; that is, of how often transcription factors bump into gates. Figure 9 shows that the oscillation frequency and amplitude remain unaffected in a large range of variation of *r* from its value in Figure 7(C)). Note that *r* is in a stochastic race against δ in *tr*, and δ is always much slower.

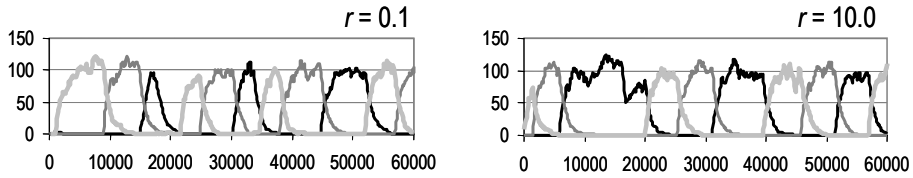


Fig. 9. Repressilator stability to changes in *r* (volume/temperature). *Cf.* Figure 7(C).

4.4 Network Properties: Fixpoint

We now discuss a network property that becomes important in later analysis. Figure 10 plots signals flowing through a sequence of *neg* gates with parameters as in Figure 7(C), except for η, the inhibition delay. On the left, the signals are alternating between high (*b,d*) and low (*a,c,e*). As η is increased, shown from left to right, the gates behave less and less like boolean operators, but the signals remain separate.

Figure 11 shows the same circuit, except for a self feedback on the head gate. With low inhibition delay η (i.e. ineffective feedback) the system is unstable (left). But soon after, as we increase η, the self feedback flattens *all* signals downstream to a common low level (middle). The signals remain at a common level over a wide range of η, although this level is raised by increasing η (right).

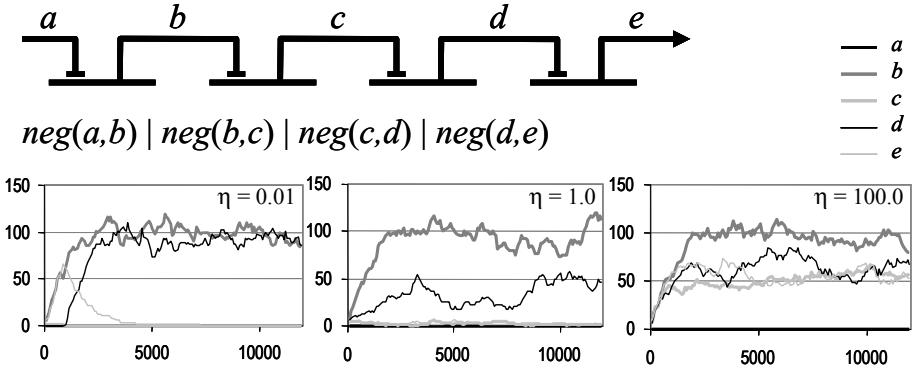


Fig. 10. A sequence of *neg* gates with three settings of their η parameter

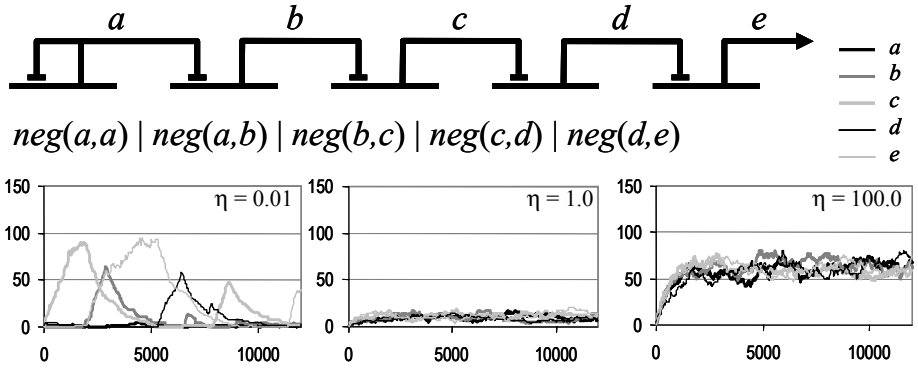


Fig. 11. The effect of head feedback on a sequence of *neg* gates

This behavior is self-regulating, and can be explained as follows. The head feedback naturally finds a fixpoint where gate input equals gate output (unless it oscillates). If the next gate has the same parameters, its output will then also equal its input, and so on down the line: all the gates will be at the same fixpoint. Different values of η and different gate response profiles may change the fixpoint level, but not its fundamental stability.

4.5 Combinatorial Circuits

As examples of non-trivial combinatorial networks and their stochastic simulation, we now examine the artificial gene circuits described by Guet *et al.* [1]. Most of those circuits are simple combinations of inhibitory gates exhibiting expected behavior. However, it was found that in some of the circuits subtle (and partially still not understood) behavior arises; we focus particularly on two of these cases.

In order to build up the different combinatorial networks easily, we begin with a version of the *neg* gate that is more flexibly parameterizable. We call it *negp*, and it has the property that, if *s* represents the rates used in the *neg* gate, then *negp*(*a*,*s*,*tr*(*b*)) = *neg*(*a*,*b*), hence *neg* is a special case of *negp*. The rates for inhibition and constitutive translation are passed as a pair *s*=(ε,η), in the second parameter. The third parameter fully encapsulates the gate product, so the gate logic is independent of it[1].

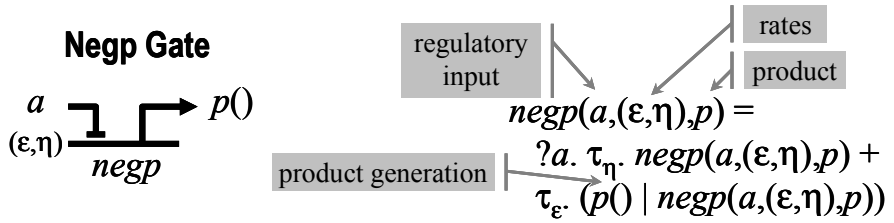


Fig. 12. A *neg* gate with parametric product *p*

[1] More technically, if we set *pb*() = *tr*(*b*) (*pb* is the process that when invoked with no arguments, invokes *tr* with argument *b*), then we have *negp*(*a*,*s*,*pb*) = *neg*(*a*,*b*); we write *negp*(*a*,*s*,*tr*(*b*)) as an abbreviation, skipping the intermediate definition of *pb*.

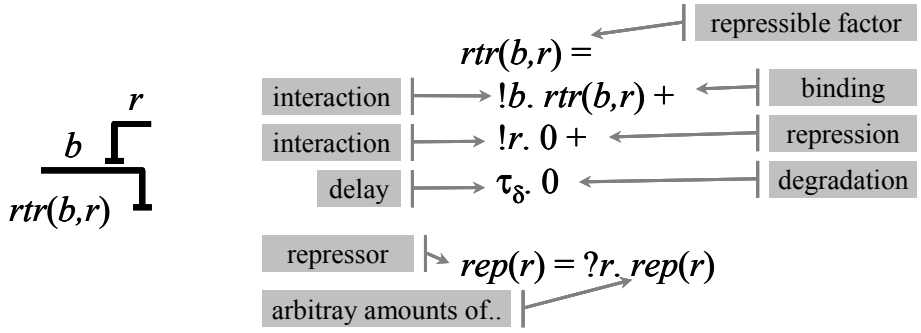


Fig. 13. Repressible transcription factors

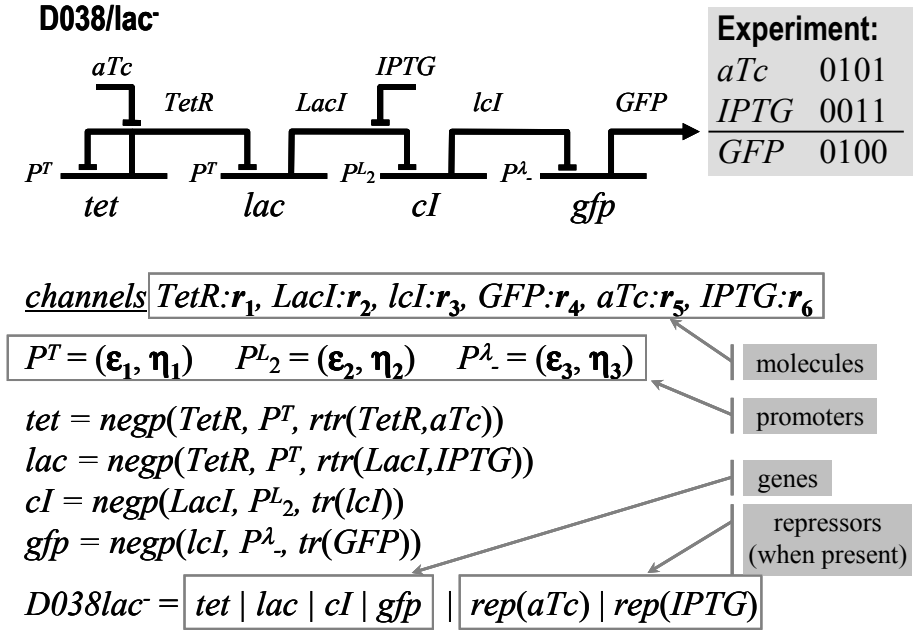


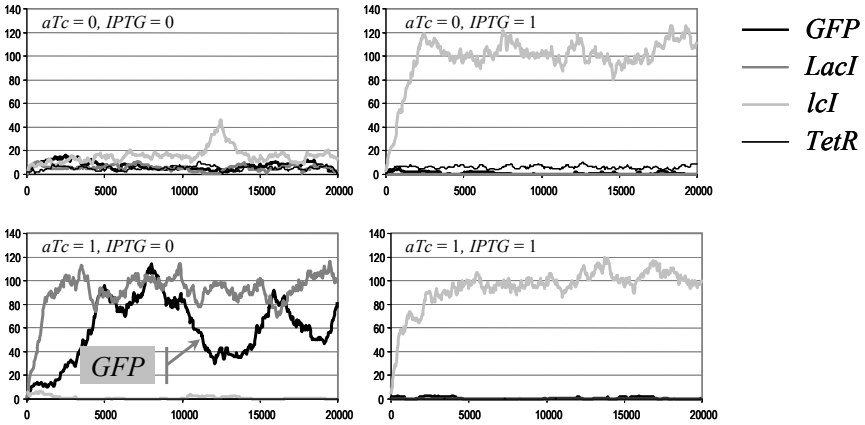
Fig. 14. D038

In addition to the old transcription factors *tr*(*b*), binding to a site *b*, we now need also transcription factors that can be repressed: *rtr*(*b*,*r*). These have three possible behaviors: binding to a site *b*, being neutralized via a site *r*, and degrading. The repression is performed by a process *rep*(*r*) that, if present, "inexhaustibly" offers ?*r*.

In the artificial gene circuits by Guet et al, the circuits are probed by varying two inputs: two so-called "inducer" proteins in the environment, aTc and IPTG, which bind specifically to the gene in question. The output of the gene circuit is detected by a reporter gene which produces a green-fluorescent protein (*GFP*) which can be optically detected.

We can now describe the circuits from [1] by simple combinations of *negp*, *tr*, *rtr*, and *rep* components. All the other names appearing here, such as *TetR, aTc*, etc., which glue the network together, are just channel names used in complementary input and output actions.

Intuitive Boolean analysis of one of the still controversial circuits, D038, in Figure 14 would suggest either oscillation (*GFP*=0.5 on average), or *GFP*=1, contrary to experiment[2]. Thus, for the given construction, a different explanation is needed. The fixpoint effect, however, which we have described in Section 4.4, does suggest an explanation for the output in the absence of repressors, whereby all signals including the output signal *GFP* are driven to a fixpoint with a low value. The addition of *GFP* renders that state unstable and drives *TetR* to 0, and hence *GFP* to 1. In all cases, the addition of *IPTG* drives *LacI* to 0 and hence *GFP* to 0. Figure 15 shows the simulation results of this system for the different values of *aTc* and *IPTG*. In circuit D038 we have thus found an example in which the modelling of the stochastic gate behaviour can indeed help to find an explanation of the observed dynamics.



$r_{1..6} = 1.0,\ \delta = 0.001$
$\varepsilon_{1,2,3} = 0.1,\ \eta_1 = 0.25\ (P^T),\ \eta_{2,3} = 1.0\ (P^L{}_2,\ P^\lambda{}_-)$

Fig. 15. D038 simulations

[2] In absence of repressors, the experimentally observed *GFP* is 0 (meaning no detectable signal), hence, by tracing boolean gates backwards, *lcI*=1, and *LacI*=0, and *TetR*=1. But by self-loop *TetR*=1 implies *TerR*=0, so the whole circuit, including *GFP* should be oscillating and averaging *GFP*=0.5. As an alternative analysis, consider the level of *TetR* (which is difficult to predict because it is the result of a negative self-feedback loop). Whatever that level is, and whether or not *aTc* is present, it must equally influence the *tet* and *lac* genes, since the promoters are the same (P^T). The option, *TetR*=*LacI*=1 gives *GFP*=1. Suppose instead *TetR*=*LacI*=0, then *lcI*=1, and *GFP*=0 as observed. But in that situation, with *TetR*=0, *aTc* should have no influence, since it can only reduce the level of *TetR*. Instead, *aTc* somehow pushes *GFP* to 1.

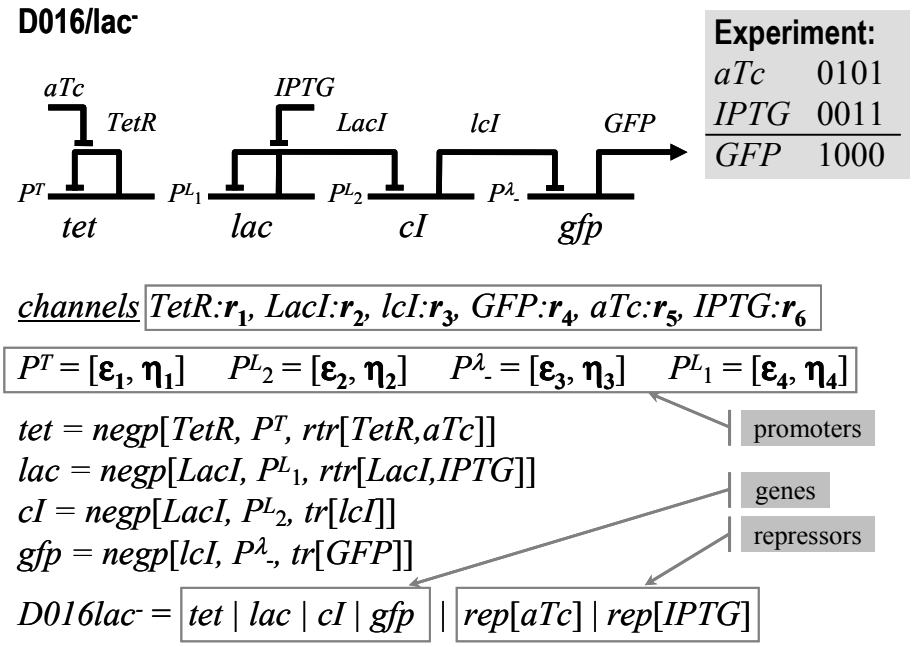


Fig. 16. D016

In a very similar fashion we can code another peculiar circuit, D016, shown in Figure 16. This circuit is perplexing because addition of *aTc*, affecting an apparently disconnected part of the circuit, changes the *GFP* output. In [18] it is suggested that this may be caused by an overloading of the degradation machinery, due to an overproduction of *TetR* when *aTc* is present, which might decrease the degradation rate of the other proteins. But even in absence of *aTc* and *IPTG*, it is surprising that *GFP* is high (about 50% of max [16]): this seems to contradict both simple boolean analysis and our fixpoint explanation which worked well for D038.

One way to rationalize the behaviour displayed by this circuit is to assume that the P^L_1-*lac* gate is operating in a region in parameter space in which the circuit dynamics is unstable. A closer examination of the instability region of our basic fixpoint circuit (Figure 10 bottom left) shows that, while the first signals in the sequence (*a*,*b*) are kept low, the subsequent signals (*c*, corresponding to *GFP* in D016, and *d,e*) all spike frequently. This may give the appearance, on the average, of high levels of *GFP*, matching the first column of the D016 experiment. Moreover, in the instability region the system responds very sensitively to changes in degradation levels: *GFP* levels can be brought down both by increasing degradation by a factor of 5 (because this brings the circuit back into the fixpoint regime) or by decreasing degradation by a factor of 1000 (so that there are enough transcription factors to inhibit all gates). In Figure 17 we begin by placing D016 in the instability regime, with *GFP* spiking (A). Then, adding *aTc* while reducing degradation suppresses all signals (B). Adding *IPTG* results in no *GFP* (C,D); moreover, reduced degradation causes overproduction (D). Even increased degradation (E) can result in no *GFP*.

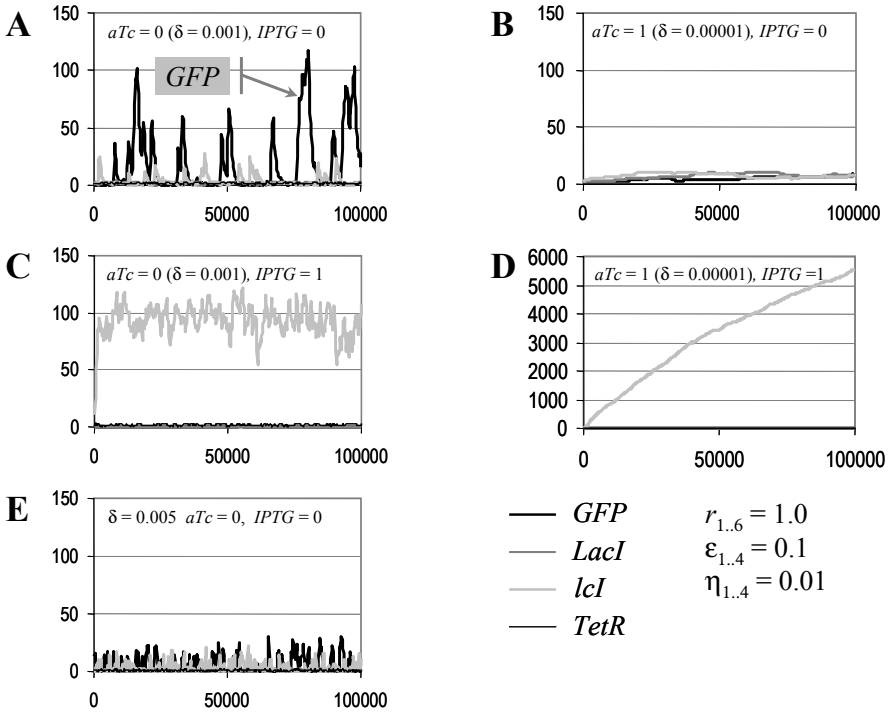


Fig. 17. D016 simulations

While a proper biological explanation of the behavior of D016 has not been obtained yet, the type of analysis we have performed here already shows the potential of the information gain from a proper study of the stochastic dynamics of the gene circuits, in particular in the case where head feedbacks are present; other authors have noted the possibility of surprises in such cases [19].

5 Conclusions

In this paper we have demonstrated how stochastic simulations of gene circuits can be built in a compositional way by employing the stochastic π-calculus. For this, we chose as a descriptive level not the molecular constituents, but rather considered each gene as a gate with corresponding inputs and outputs. On this level, compositionality is illustrated, for example, by our treatment of the repressilator circuit: the definition of the *neg* gate could be left unchanged when the definition of the transcription factor *tr* was refined. Our approach is mechanistic in the sense that we (re-)construct a biological system from discrete elements and then deduce the system behaviour as arising from the interactions of the components. This differs from modelling attempts of the same systems in the bioinformatics literature which only looked at gene expression levels without considering their origin [18]. Our approach, while being abstract, is advantageous as it allows a considerable flexibility in the level of detail

with which components and their interactions are described (see the Appendix for further illustration). While the adopted level of the description may be considered coarse and qualitative, the π-calculus approach easily allows for refinements (i.e., inclusion of additional detail down to molecular levels of description) to match available knowledge.

Apart from these analytical and conceptual advantages in building up the different circuits, we stress that the ease of use of the compositional approach in combination with stochastic simulations is particularly useful for hypothesis testing. It can build on available knowledge, but the outcome of the stochastic simulations of the interacting components yields a highly non-trivial check of expectations. By comparison, Boolean analysis or intuitive ideas are obviously too naïve and thus can easily be misleading.

The sensitivity of the gene network dynamics to parameter choice has to be contrasted with the lack of quantitative knowledge of promoter strengths, or even qualitative relationships between the different promoters [19]. In the absence of "true" (i.e., experimentally validated) parameter values, a detailed analysis of the stochastic behaviour of the gene networks resulting from a systematic parameter variation can be a very useful - but clearly not sufficient - step to avoid misinterpretations of experiments.

To conclude, we believe that the compositional approach we propose for the formulation of stochastic models of gene networks will allow a useful path for more detailed, quantitative studies of regulatory mechanisms, and in particular for the testing of hypotheses of complex system behavior. It may be considered as one step towards the development of flexible languages and simulation tools for computational biology, for which a need has recently been expressed by several biologists ([20]-[22]).

References

[1] Guet, C.C., Elowitz, M.B., Hsing, W. & Leibler, S. (2002) Combinatorial synthesis of genetic networks. Science 296 1466-1470.

[2] Thattai, M. & van Oudenaarden, A. (2001) Intrinsic noise in gene regulatory networks. Proc. Nat. Acad. Sci. 98, 8614- 8619.

[3] Paulsson, J., Berg, O.G. & Ehrenberg M. (2000) Stochastic Focusing: fluctuation-enhanced sensitivity of intracellular regulation. Proc. Nat. Acad. Sci. 97, 7148-7153.

[4] Milner, R. (1999) Communicating and Mobile Systems: The π-Calculus. Cambridge University Press.

[5] Priami, C., Regev, A., Shapiro, E. & Silverman, W. (2001) Application of stochastic process algebras to bioinformatics of molecular processes. Information Processing Letters 80 25-31.

[6] Regev, A. (2002) Computational Systems Biology: A Calculus for Biomolecular knowledge. Ph.D. Thesis, Tel Aviv University.

[7] Regev, A. & Shapiro, E. (2002) Cellular abstractions: Cells as computation. Nature 419 343.

[8] Regev, A., Panina, E.M., Silverman, W., Cardelli, L. & Shapiro, E. (2004) BioAmbients: An abstraction for biological compartments. Theoretical Computer Science, 325(1) 141-167.

[9] Cardelli, L. (2004) Brane Calculi - Interactions of Biological Membranes. Computational Methods in Systems Biology. Springer. 257-278.
[10] Chiarugi, D., Curti, M., Degano, P. & Marangoni, R.: VICE: A VIrtual CEll. CMSB 2004: 207-220.
[11] Kuttler, C.& Niehren, J. (2005) Gene Regulation in the Pi Calculus: Simulating Cooperativity at the Lambda Switch. Transactions on Computational Systems Biology, to appear.
[12] Elowitz, M.B., Leibler. S. (2000) A synthetic oscillatory network of transcriptional regulators. Nature 403 335-338.
[13] Philips, A. & Cardelli, L., (2005). A Correct Abstract Machine for the Stochastic Pi-calculus. Proc. BioConcur 2004.
[14] Gillespie, D. (1977) Exact stochastic simulation of coupled chemical reactions. J. Chem. Phys. 81 2340-2361.
[15] Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, 1996.
[16] Guet, C.C., personal communication.
[17] Wigler, M. & Mishra, B. (2002) Wild by nature. Science 296 1407-1408.
[18] Mao, L. & Resat, H. (2004) Probabilistic representation of gene regulatory networks. Bioinformatics 20 2258-2269.
[19] Ronen, M., Rosenberg, R., Shraiman, B.I. & Alon, U. (2002) Assigning numbers to the arrows: Parameterizing a gene regulation network by using accurate expression kinetics. Proc. Nat. Acad. Sci. 99 10555-10560.
[20] Brenner, S. (1995) Loose Ends. Curr. Biology 5 332.
[21] Bray, D. (2001) Reasoning for Results. Nature 412 863.
[22] Lazebnik, Y. (2002) Can a biologist fix a radio? Or, what I learned while studying apoptosis. Cancer Cell 2 179-182.

Appendix

A Simulator for the Stochastic π-Calculus

The following is a detailed description of the Stochastic π-calculus and the Stochastic Pi Machine, as presented in [13].

$P,Q ::=$	new x P	Restriction	$\Sigma ::=$	$\mathbf{0}$	Null
|	$P \mid Q$	Parallel	|	$\pi.P + \Sigma$	Action
|	Σ	Choice	$\pi ::=$	$!x(n)$	Output
|	$*\pi.P$	Replication	|	$?x(m)$	Input

Def. 1. Syntax of the Stochastic π-calculus

$$!x(n).P + \Sigma \mid ?x(m).Q + \Sigma' \xrightarrow{rate(x)} P \mid Q_{\{n/m\}} \qquad [1]$$

$$P \xrightarrow{r} P' \Rightarrow P \mid Q \xrightarrow{r} P' \mid Q \qquad [2]$$

$$P \xrightarrow{r} P' \Rightarrow \text{new } x\ P \xrightarrow{r} \text{new } x\ P' \qquad [3]$$

$$Q \equiv P \xrightarrow{r} P' \equiv Q' \Rightarrow Q \xrightarrow{r} Q' \qquad [4]$$

Def. 2. Reduction in the Stochastic π-calculus

Stochastic π-calculus. A biological system can be modeled in the stochastic π-calculus by representing each component of the system as a calculus process P that precisely describes what the component can do. According to Def. 1, the most basic component is a choice Σ between zero or more output $!x(n)$ or input $?x(m)$ actions that the component can perform. Two components P and Q can be combined together using parallel composition $P|Q$, and a component P can be given a private interaction channel x using restriction *new x P*. In addition, multiple copies of a given component $\pi.P$ can be cloned using replication $*\pi.P$. Standard syntax abbreviations are used, such as writing π for $\pi.\mathbf{0}$ and $\pi.P$ for $\pi.P + \mathbf{0}$.

Two components in a biological system can interact by performing complementary input and output actions on a common channel. During such an interaction, the two components can also exchange information by communicating values over the channel. Each channel x is associated with a corresponding interaction rate given by $rate(x)$ and the interaction between components is defined using reduction rules of the form $P \xrightarrow{r} P'$. Each rule of this form describes how a process P can evolve to P' by performing an interaction with rate r. According to Def. 2, a choice containing an output $!x(n).P$ can interact with a parallel choice containing an input $?x(m).Q$. The interaction occurs with $rate(x)$, after which the value n is assigned to m in process Q (written $Q_{\{n/m\}}$) and processes P and $Q_{\{n/m\}}$ are executed in parallel (Eq. 1). Components can also interact in parallel with other components (Eq. 2) or inside the scope of a private channel (Eq. 3), and interactions can occur up to re-ordering of components (Eq. 4), where $P \equiv Q$ means that the component P can be re-ordered to match the component Q. In particular, the re-ordering $*\pi.P \equiv \pi.(P \mid *\pi.P)$ allows a replicated input $*?x(m).Q$ to clone a new copy of Q by reacting with an output $!x(n).P$.

V,U::=	new x V	Restriction	A,B::=	[]	Empty
	A	List		Σ:: A	Choice

Def. 3. Syntax of the Stochastic Pi Machine

$$\begin{array}{l} x,\tau = Gillespie(A) \\ \wedge\ A > (?x(m).P + \Sigma) :: A' \\ \wedge\ A' > (!x(n).Q + \Sigma') :: A'' \end{array} \Rightarrow A \xrightarrow{rate(x)} P_{\{n/m\}} : Q : A'' \qquad [5]$$

$$V \xrightarrow{r} V' \Rightarrow \text{new } x\ V \xrightarrow{r} \text{new } x\ V' \qquad [6]$$

Def. 4. Reduction in the Stochastic Pi Machine

Stochastic Pi Machine. The Stochastic Pi Machine is a formal description of how a process of the stochastic π-calculus can be simulated. A given process P is simulated by first encoding the process to a corresponding simulator term V, consisting of a list of choices with a number of private channels:

$$\text{new } x_1 \text{ ... new } x_N \ (\Sigma_1::\Sigma_2::...::\Sigma_M::[])$$

This term is then simulated in steps, according to the reduction rules in Def. 4. A list of choices A is simulated by first using a function *Gillespie*(A) to stochastically determine the next interaction channel x and the corresponding interaction time τ. Once an interaction channel x has been chosen, the simulator uses a *selection operator* (>) to randomly select a choice $?x(m).P + \Sigma$ containing an input on channel x and a second choice $!x(n).Q + \Sigma'$ containing an output on x. The selected components can then interact by synchronizing on channel x, where the value n is sent over channel x and assigned to m in process P (written $P_{\{n/m\}}$). After the interaction, the unused choices Σ and Σ' are discarded and the processes $P_{\{n/m\}}$ and Q are added to the remainder of the list to be simulated, using a construction operator (:) (Eq. 5). An interaction can also occur inside the scope of a private channel (Eq. 6). The simulator continues performing interactions in this way until no more interactions are possible.

The function *Gillespie*(A) is based on the Gillespie Algorithm [14], which uses a notion of *channel activity* to stochastically choose a reaction channel from a set of available channels. The activity of a channel corresponds to the number of possible combinations of reactants on the channel. Channels with a high activity and a fast reaction rate have a higher probability of being selected. A similar notion of activity is defined for the Stochastic Pi Machine, where $Act_x(A)$ denotes the number of possible combinations of inputs and outputs on channel x in A:

$$Act_x(A) = In_x(A) \times Out_x(A) \ - \ Mix_x(A)$$

$In_x(A)$ and $Out_x(A)$ are defined as the number of available inputs and outputs on channel x in A, respectively, and $Mix_x(A)$ is the sum of $In_x(\Sigma_i) \times Out_x(\Sigma_i)$ for each choice Σ_i in A. The formula takes into account the fact that an input and an output in the same choice cannot interact, by subtracting $Mix_x(A)$ from the product of the number of inputs and outputs on x. Once the values x and τ have been calculated, the simulator increments the simulation time by delay τ and uses the selection operator to randomly choose one of the available interactions on x according to (Eq. 5). This is achieved by randomly choosing a number $n \in [1..In_x(A)]$ and selecting the nth input in A, followed by randomly selecting an output from the remaining list in a similar fashion. The application of the Gillespie algorithm to the Stochastic Pi Machine is summarized in Def. 3, where $fn(A)$ denotes the set of all channels in A.

---

1. For all $x \in fn(A)$ calculate $a_x = Act_x(A) \times rate(x)$
2. Store non-zero values of a_x in a list (x_μ, a_μ), where $\mu \in 1...M$.
3. Calculate $a_0 = \sum_{\nu=0}^{M} a_\nu$

4. Generate two random numbers $n_1, n_2 \in [0,1]$ and calculate τ,μ such that:

$$\tau = (1/a_0)\ln(1/n_1)$$

$$\sum_{\nu=1}^{\mu-1} a_\nu < n_2 a_0 \leq \sum_{\nu=1}^{\mu} a_\nu$$

5. $Gillespie(A) = (x_\mu, \tau)$.

Def. 5. Calculating *Gillespie(A)* according to (13)

For improved efficiency, the simulator can be modified to store a list of values for each channel x in A, of the form:

$$x, In_x(A), Out_x(A), Mix_x(A), a_x$$

After each reduction has been performed, it is only necessary to update the values for those channels that were affected by the reduction, and then use Def. 5 on the updated values to choose the next reaction channel and calculate the delay.

To gain confidence in our simulation technique, we have conducted detailed simulations of the model chemical systems which were simulated in [14] using the Gillespie algorithm. Comparable results were obtained by modeling each system as a π-calculus process and simulating the resulting processes in the Stochastic Pi Machine.

Repressilator Code

From the simple examples discussed previously, the structure of the SPiM programs should now be clear. The following is the complete code for the repressilator simulation in Figure 7(C) of the paper, for the SPiM simulator (v0.04). In order to clarify parts of the code, comments are added in (* ... *) brackets.

```
(* Simulation time, samples, and plotting  *)
directive sample 90000.0 500
directive plot !a as "a"; !b as "b"; !c as "c"

(* Parameters *)
val dk = 0.001           (* Decay rate *)
val inh = 0.001          (* Inhibition rate *)
val cst = 0.1            (* Constitutive rate *)
val bnd = 1.0            (* Protein binding rate *)

(* Transcription factor  *)
let tr(p:chan()) =
  do !p; tr(p)
  or delay@dk
```

```
(* Neg gate *)
let neg(a:chan(), b:chan()) =
  do ?a; delay@inh; neg(a,b)
  or delay@cst; (tr(b) | neg(a,b))

(* The circuit *)
new a @ bnd: chan()
new b @ bnd: chan()
new c @ bnd: chan()

run (neg(c,a) | neg(a,b) | neg(b,c))
```

D038,D016 Code

The following is the complete code for the of the D038 and D016 simulations in Figure 15 and Figure 17, for the SPiM simulator (v0.04).

```
(* Simulation time, samples, and plotting  *)
directive sample 20000.0 500
directive plot !GFP as "GFP"; !LacI as "LacI";
          !LambcI as "LambcI"; !TetR as "TetR"

(* Degradation rate *)
val dk = 0.001
(* val dk = 0.00001    for D016 when aTc is present *)

(* Transcription factor *)
let tr(b:chan()) =
  do !b; tr(b)
  or delay@dk

(* Repressible transcription factor *)
let rtr(b:chan(), r:chan()) =
  do !b; rtr(b,r)
  or !r
  or delay@dk

(* Repressor *)
let rep(r:chan()) =
  ?r; rep(r)

(* Negp gate *)
let negp(a:chan(), (cst:float, inh:float), p:proc()) =
  do ?a; delay@inh; negp(a,(cst,inh),p)
  or delay@cst; (p() | negp(a,(cst,inh),p))

(* Wiring *)
new TetR @1.0: chan()            (* TetR protein *)
new LacI @1.0: chan()            (* LacI protein *)
new LambcI @1.0: chan()   (* LambcI protein *)
new GFP @1.0: chan()             (* GFP protein *)
new aTc @100.0: chan()           (* aTc inducer *)
new IPTG @100.0: chan()   (* IPTG inducer *)

(* Auxiliary definitions: negp products *)
let rtr_TetR_aTc() = rtr(TetR,aTc)
let rtr_LacI_IPTG() = rtr(LacI,IPTG)
```

```
let tr_LambcI() = tr(LambcI)
let tr_GFP() = tr(GFP)

(* D038 Circuit *)
val PT = (0.1, 0.25)   (* PT constitutive and inhibition rates *)
val PL2 = (0.1, 1.0)   (* PL2 constitutive and inhibition rates *)
val Plm = (0.1, 1.0)   (* Plm constitutive and inhibition rates *)

let tet() = negp(TetR, PT, rtr_TetR_aTc)
let lac() = negp(TetR, PT, rtr_LacI_IPTG)
let cI()  = negp(LacI, PL2, tr_LambcI)
let gfp() = negp(LambcI, Plm, tr_GFP)

run
( tet() | lac() | cI() | gfp()
(* | rep(aTc)     uncomment to test with aTc *)
(* | rep(IPTG)    uncomment to test with IPTG *)
)
(* D016 Circuit *)
val PT = (0.1, 0.01)   (* PT constitutive and inhibition rates *)
val PL1 = (0.1, 0.01)   (* PL1 constitutive and inhibition rates *)
val PL2 = (0.1, 0.01)   (* PL2 constitutive and inhibition rates *)
val Plm = (0.1, 0.01)   (* Plm constitutive and inhibition rates *)

let tet() = negp(TetR, PT, rtr_TetR_aTc)
let lac() = negp(LacI, PL1, rtr_LacI_IPTG)
let cI()  = negp(LacI, PL2, tr_LambcI)
let gfp() = negp(LambcI, Plm, tr_GFP)

run
( tet() | lac() | cI() | gfp()
(* | rep(aTc)     uncomment to test with aTc *)
(* | rep(IPTG)    uncomment to test with IPTG *))
```

Complexation

Complexation can be modeled in stochastic process calculi by using a technique originally developed by Aviv Regev and Ehud Shapiro [6][7]. This technique provides a simple illustration of a major feature of process calculi that we have not emphasized in the main text: the dynamic creation of fresh communication channels. A fresh (unique) channel can be dynamically created, operationally, by incrementing a global counter, or by picking a random number. Process calculi abstract from these operational details by a formalized notion of what it means for a channel to be *fresh*. The operator *new* c_r; P creates a fresh channel named c with rate r for use in P (distinct from any other channel that might also be named c).

We want to model two proteins P and Q that combine into a complex $P{:}Q$ at some rate r, and break apart again at some rate s. Let cx denote the complexation interaction of the two proteins: this is modeled as a single "public" channel cx of rate r, where multiple copies of P and Q can interact to come together and form complexes. Let dx denote the decomplexation interaction of two bound proteins: this is modeled as a separate channel dx of rate s for each complex. Such a fresh channel is established separately for each complex at the time of complexation, for the purpose of subsequently breaking up.

$P = new\ dx_s\ !cx(dx);\ !dx;\ P$
$Q = ?cx(x);\ ?x;\ Q$ where x is an input variable

If we consider just one copy of P and one of Q, for simplicity, the initial system $P|Q$ consisting of two separate proteins can evolve by P creating a fresh channel dx and outputting this dx over the public channel cx, where it can be input by Q and bound to its input variable x. At this point the system has evolved into the configuration $new\ dx_s\ (!dx;\ P) \mid (?dx;\ Q)$, where dx is unknown to any other actual or potential process in the system. This state represents the complex of the original P and Q. Next, an interaction can happen over this particular dx channel among the only two processes that share it: this is the decomplexation event resulting in the initial state $P|Q$.

$P|Q \rightarrow_r new\ dx_s\ (!dx;\ P) \mid (?dx;\ Q) \rightarrow_s P|Q$ where dx is fresh

Many variations on this theme are possible, including modeling the binding, unbinding, and cooperative binding of transcription factors.

Neg Gate Dynamic Response Profile

We test the dynamic response profile of the *neg* gate of Figure 3. To observe some of its behavior under operating conditions, we provide an input consisting of a signal raising linearly from 0 to 100, and then falling linearly from 100 to 0. That means 100 copies of input molecules, where each molecule is injected at a certain time and can interact or decay a certain number of times (thus shaping the input curve).

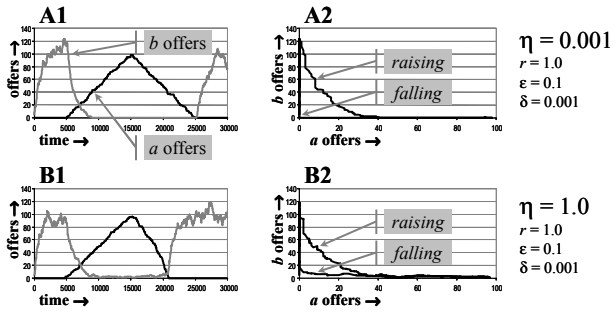


Fig. 18. Neg Gate Response Profile

Initially, in absence of any input, the output of the *neg* gate quickly raises to about 100. As the input signal ramps up, the output signal decays, and as the signal ramps down the output rises again, but with an asymmetric profile. (Figure 18 (A1,B1): the ramping down of the input signal in B1 appears abbreviated because the signal is consumed at a higher rate by the gate.) Plotting input vs output for the same data (Figure 18 (A2,B2)) we can see a roughly hyperbolic response with two distinct curves corresponding to raising and falling inputs. We show the plots for a highly sensitive ("Boolean") gate with η=0.001 (Figure 18 (A1,A2)) and a less sensitive gate with η=1.0 (Figure 18 (B1,B2)); these parameters cover the range used in simulations

in the main text. As in the main text, what is actually plotted is the number of (output) communication *offers* on the channels.

These response profiles illustrate the fact that, e.g., in the repressilator, each signal dynamically shapes the next signal and is shaped by the intake of the next gate.

The following is the complete code used to obtain the graphs, for the SPiM simulator (v0.04).

```
(* Simulation time, samples, and plotting  *)
directive sample 30000.0 1000
directive plot !a as "a"; !b as "b"

(* Parameters *)
val dk  = 0.001   (* Output protein decay rate *)
val inh = 0.001   (* Inhibition rate, or 1.0 *)
val cst = 0.1     (* Constitutive rate *)
val bnd = 1.0     (* Protein binding rate *)

(* Transcription factor *)
let tr(p: chan()) = do !p;tr(p) or delay@dk

(* Neg gate *)
let neg(a:chan(), b:chan()) =
  do ?a; delay@inh; neg(a,b)
  or delay@cst; (tr(b) | neg(a,b))

(* Probe signal: linearly raising and falling *)
val pbdk = 0.1    (* Probe signal decay rate *)
let probe1(p:chan(),n:int) =
  if n=0 then ()
  else (do !p;probe1(p,n-1) or delay@pbdk; probe1(p,n-1))
let dprobe1(p:chan(),d:int,n:int) =
  if d=0 then probe1(p,2*10*n)
  else delay@pbdk;dprobe1(p,d-1,n)
let probe(p:chan(),m:int) =
  if m=0 then ()
  else (dprobe1(p,500+(10*m),100-m) | probe(p,m-1))

(* Probing *)
new a@bnd:chan()  new b@bnd:chan()
run (neg(a,b) | probe(a,100))
```