A compositional approach to the stochastic dynamics of gene networks – Supplementary Material

Ralf Blossey*, Luca Cardelli†, and Andrew Phillips†

*Interdisciplinary Research Institute, F-59652 Villeneuve d'Ascq, France; †Microsoft Research, CB3 0FB Cambridge, United Kingdom

A Simulator for the Stochastic Pi-calculus

The following is a detailed description of the Stochastic Pi-Calculus and the Stochastic Pi Machine, as presented in (13). All references in this Supplementary Material refer to the main paper.

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

Def. 1. Syntax of the Stochastic Pi-calculus

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

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

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

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

Def. 2. Reduction in the Stochastic Pi-calculus

Stochastic Pi-calculus. A biological system can be modeled in the stochastic pi-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 $!x(n).P$.

$V,U ::=$	$\text{new } x\ V$	Restriction	$A,B ::=$	$[]$	Empty
	A	List		$\Sigma :: A$	Choice

Def. 3. Syntax of the Stochastic Pi Machine

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

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

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 pi-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:

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

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. 5, 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 (14)

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

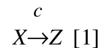
$$x,In_x(A),Out_x(A),Mix_x(A),a_x$$

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.

Gillespie Test Cases

This section describes how a number of spatially homogeneous model chemical systems can be simulated in the Stochastic Pi Machine. Each of the systems presented here was previously defined as a set of reaction equations, which were simulated in (14) using the Gillespie algorithm. This section describes how comparable results can be obtained by modeling each system as a pi-calculus process and simulating the resulting processes in the Stochastic Pi Machine. For further details on the models and the references to the original literature, see (14).

Radioactive Decay. One of the simplest systems that can be simulated is the irreversible isomerization reaction, commonly referred to as radioactive decay. In this system, a species of molecule X decays with rate c to a species Z:



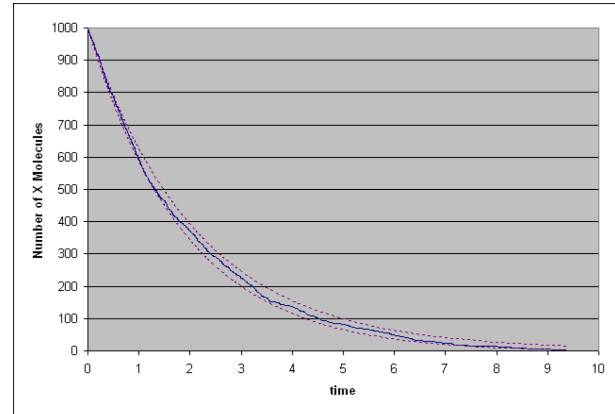
$$X \xrightarrow{c} Z \quad [1]$$

This can be modeled in the stochastic pi-calculus as a process $X()$, which performs a stochastic delay τ_c with rate c and then executes the process $Z()$:

$$X() = \tau_c.Z() \quad [2]$$

This system was simulated up to time t=10, with c=0.5 and an initial number of X molecules X_0=1000. The number of X molecules was then plotted versus time. The SPiM code for this simulation is given in Fig. S1, together with the corresponding simulation results. Note that since the process $Z()$ does not participate in any reactions it can be omitted from the pi-calculus model. The correspondence between the SPiM code and the more compact syntax used in the main paper should be obvious, knowing the following: "do P or Q" stands for "P + Q", "delay@r" stands for "τ_r", "val" introduces constant definitions, "let" introduces process definitions, and "new a@r : chan" introduces a new channel named "a" with rate "r".

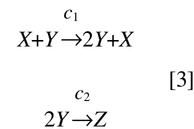
In this simple example, it is possible to solve analytically the stochastic formulation of Eq. 1 and calculate the mean and rms deviation. It turns out that the stochastic mean $X^{(1)}(t) = X_0e^{-ct}$ and the deviation $\Delta(t) = (X_0e^{-ct}(1-e^{-ct}))^{1/2}$. The two-standard deviation envelope, defined as $X^{(1)}(t) \pm \Delta(t)$, was superimposed on the simulation results for Fig. S1 in order to compare them with the predictions of the stochastic formulation. One can observe that the stochastic fluctuations of a given simulation generally lie within the boundaries of the two-standard deviation envelope.




```
directive sample 10.0
directive plot X()
val c = 0.5

let X() = delay@c
run 1000 of X()
```

Fig. S1. SPiM code and simulation results for Eq. 2 with c=0.5 and X_0=1000. The two-standard deviation envelope (dotted) has been calculated from the stochastic formulation of Eq. 1 and superimposed on the results.

Malek-Mansour-Nicolis reaction. The following system of reactions was once proposed as a refutation of the basic stochastic hypothesis:



$$X+Y \xrightarrow{c_1} 2Y+X$$
$$2Y \xrightarrow{c_2} Z \quad [3]$$

In particular, Malek-Mansour and Nicolis showed that the stochastic formulation of this system based on a Master equation has only a single steady-state solution at Y=0, while the

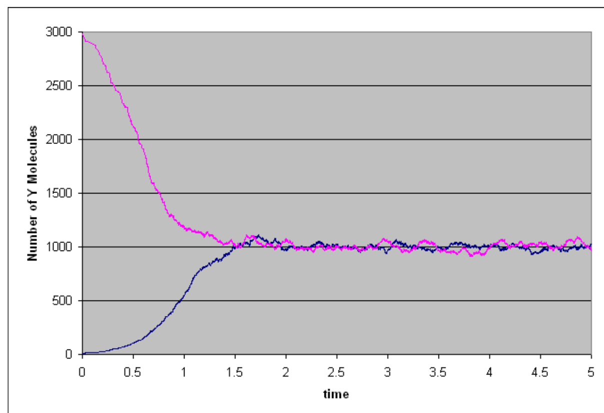
deterministic formulation has *two* steady-state solutions, an unstable one at Y=0 and a stable one at $Y=c_1X/c_2$. As a result, they concluded that the stochastic formulation destroys the stable solution of the deterministic formulation, and preserves only the trivial unstable solution. They hypothesized that even if the system is started with a large number of Y molecules it will eventually reach a steady state at Y=0, in apparent contradiction with the deterministic formulation.

In order to check this hypothesis, the system of Eq. 3 can be modeled as a pi-calculus process and simulated in SPiM. Each X molecule is modeled as a process $X()$, which can perform an input on channel c_1 and remain as $X()$. Each Y molecule is modeled as a process $Y()$, which can either perform an output on c_1 and evolve to two parallel copies of $Y()$, or perform an input on c_2 and evolve to $Z()$, or perform an output on c_2.

$$\begin{aligned} X() &= ?c_1.X() \\ Y() &= !c_1.(Y() \mid Y()) \; + \; ?c_2.Z() \; + \; !c_2 \end{aligned} \qquad [4]$$

The input and output on c_2 are used to model the fact that two Y molecules can interact with each other to produce a Z molecule. In this model, a given pair of Y molecules can interact in two possible ways: either the first Y molecule can perform an input on c_2 and the second molecule can perform an output on c_2, or vice-versa. As a result, the rate of channel c_2 needs to be adjusted so that $rate(c_2)$ in the pi-calculus model (Eq. 4) is equal to $c_2/2$ in the reaction model (Eq. 3).

The system was simulated up to time t=5, with $rate(c_1)$=5.0, $rate(c_2)$=0.0025 and an initial number of Y molecules Y_0=10. The number of Y molecules was plotted over time and the simulation was then repeated with Y_0=3000. The SPiM code for the first simulation is given in Fig. S2, together with the results for both simulations. As with the previous system, the process $Z()$ does not participate in any reactions and can be omitted from the pi-calculus model.

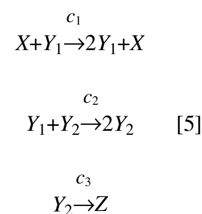



```
directive sample 5.0
directive plot Y()
new c1@5.0:chan
new c2@0.0025:chan

let X() = ?c1; X()
let Y() =
  do !c1; (Y() | Y())
  or !c2
  or ?c2
run (X() | 10 of Y())
```

Fig. S2. SPiM code and simulation results for (Eq. 4) with *rate*(c_1)=5, *rate*(c_2)=0.005 and initial values Y_0=10. The simulation results for Y_0=3000 are also given.

The simulation results show that different initial conditions of Y_0=10 and Y_0=3000 lead to a situation in which the number of Y molecules fluctuates in an apparently stable manner around the steady state value of c_1X/c_2=1000, as predicted by the deterministic formulation of Eq. 3. Although in theory the number of Y molecules will eventually reach 0 as $t \longrightarrow \infty$, in practice the system will continue to oscillate indefinitely around the steady state value of c_1X/c_2, with a very low probability of randomly fluctuating from this steady state value to Y=0. In fact, analytical calculations (14) have shown that the variance about the steady-state mean $Y_s^{(1)}$ is given by $\Delta_s^2=(3/2)Y_s^{(1)}$, which gives a standard deviation of about 39 for a steady state value of 1000. This comparison between analytical calculation and simulation results illustrates how stochastic simulations can help clarify the subtle differences between deterministic and stochastic formulations of chemical systems.

The Lotka Reactions. The Lotka reactions can be used to model a simple predator-prey ecosystem, in which a prey species Y_1 feeds on an inexhaustible food source X to reproduce, a predator species Y_2 feeds on Y_1 to reproduce and the predator species Y_2 can die of natural causes:



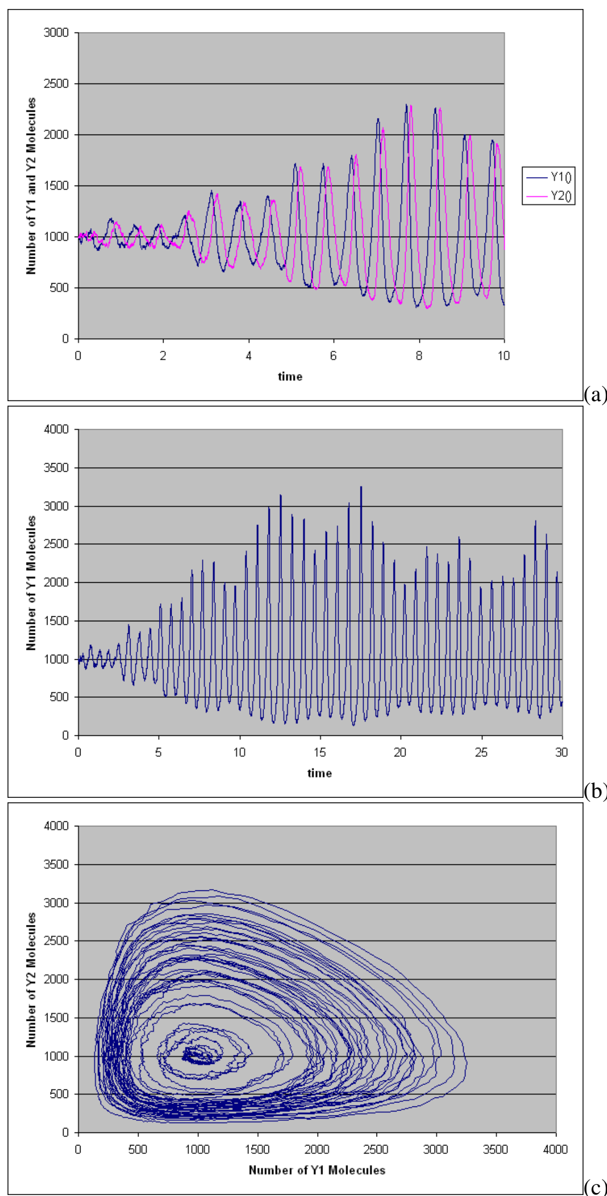
$$\begin{aligned} X+Y_1 &\xrightarrow{c_1} 2Y_1+X \\ Y_1+Y_2 &\xrightarrow{c_2} 2Y_2 \\ Y_2 &\xrightarrow{c_3} Z \end{aligned} \qquad [5]$$

This system can be given a deterministic formulation using differential equations, which can be shown to have a steady state of $Y_1=Y_{1s}=c_3/c_2$ and $Y_2=Y_{2s}=c_1X/c_2$. Therefore, if the system has initial populations $Y_1=Y_{1s}$ and $Y_2=Y_{2s}$ at time t=0, the deterministic formulation predicts that this situation will persist indefinitely.

The system of Eq. 5 can be modeled as a pi-calculus process and simulated in SPiM. The inexhaustible food source X is modeled as a process $X()$, which can be "eaten" by performing an input on channel c_1 and then remain as $X()$. The prey Y_1 is modeled as a process $Y_1()$, which can eat by performing an output on c_1 and then reproduce as two $Y_1()$ processes in parallel, or be killed by performing an input on c_2 and then disappear. The predator Y_2 is modeled as a process $Y_2()$, which can eat by performing an output on c_2 and then reproduce as two $Y_2()$ processes, or die of natural causes by performing a stochastic delay τ_{c3} and then disappear.

$$\begin{aligned} X() &= ?c_1.X() \\ Y_1() &= !c_1.(Y_1() \mid Y_1()) \; + \; ?c_2 \\ Y_2() &= !c_2.(Y_2() \mid Y_2()) \; + \; \tau_{c3} \end{aligned} \qquad [6]$$

This system was simulated up to time t=30, with $rate(c_1)$=10.0, $rate(c_2)$=0.01, c_3=10.0, initial populations $Y_1=Y_2$=1000 and an inexhaustible species X. The SPiM code for the simulation is given in Fig. S3, together with the corresponding simulation results. The results show that, instead of remaining at a constant value of 1000, the number of Y_1 and Y_2 species oscillates with a fairly stable frequency and phase, but markedly unstable amplitude. Fig. S3(a) shows how the predator population lags behind that of the prey, Fig. S3(b) shows the stability of the frequency and instability of the amplitude of the oscillations in the prey population and Fig. S3(c) shows the counter-clockwise orbits traced out in the Y_1Y_2 plane.



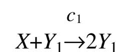

```
directive sample 30.0
directive plot Y1(); Y2()
new c1@10.0:chan
new c2@0.01:chan
val c3 = 10.0

let X() = ?c1; X()
let Y1() =
  do !c1; (Y1() | Y1())
  or ?c2
let Y2() =
  do !c2; (Y2() | Y2())
  or delay@c3
run (X() | 1000 of Y1() | 1000 of Y2())
```

Fig. S3. SPiM code and simulation results for the Lotka reactions (Eq. 6) with *rate*(c_1)=10.0, *rate*(c_2)=0.01, c_3=10.0 and initial values Y_1=Y_2=1000. Results for (a) Y_1,Y_2 vs. t with $0<t\leq10$, (b) Y_1 vs. t with $0<t\leq30$ and (c) Y_2 vs. Y_1.

The simulation results can be logically explained by the fact that a rise in the prey population provides additional food for the reproduction of the predators, resulting in a rise in predator population shortly afterwards. This in turn leads to an increase in consumption of prey species, resulting in a decline in the prey population, followed closely by a decline in predator population, and so on. The results can also be explained by analyzing the stability of the solutions of the deterministic formulation. Such analysis shows that the orbits in the Y_1Y_2 plane are *neutrally stable*, i.e. when perturbed slightly to a point (Y_{11},Y_{21}) off the orbit, the system will begin orbiting on the solution orbit that passes through the new point (Y_{11},Y_{21}). Therefore, any random fluctuations in Y_1 and Y_2 will result in the system wandering between neutrally stable orbits.

Furthermore, the wide amplitude fluctuations indicate that it is only a matter of time before the orbits intersect with either the Y_1 or Y_2 axis. Therefore, as $t\longrightarrow\infty$ either the Y_1 prey species becomes extinct and the Y_2 predator species dies out soon afterwards, or the Y_2 predator species becomes extinct and the Y_1 species tends to infinity. This contrasts with the predictions of the deterministic formulation, which suggest that the populations of predator and prey will remain constant over time. These results indicate the importance of taking into account stochastic fluctuations when trying to predict the behavior of a system.

A number of variations on the Lotka reactions (Eq. 5) can also be simulated. In particular, the food source X can be made finite by changing the definition of reaction c_1:



$$X+Y_1\xrightarrow{c_1}2Y_1$$

This can be modeled in the pi-calculus by changing the corresponding definition of process X():

$$X() = ?c_1$$

The resulting system can be simulated in SPiM by starting with a large quantity of food source X, as shown in Fig. S4. The simulation results indicate that the depletion of the food source X is more detrimental to the predator than to the prey. They show that the predators become extinct at $t\simeq21$, after which the remaining food source X is consumed by the prey for reproduction.

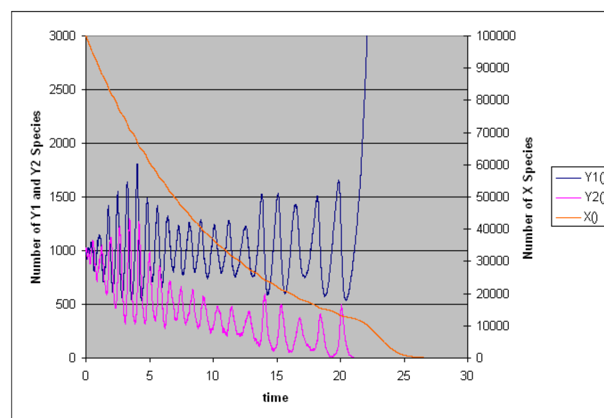
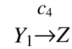


Fig. S4. Simulation results for the Lotka reactions (Eq. 6) but with limited number of X species. Simulation up to time t=30, with *rate*(c_1)=0.0001, *rate*(c_2)=0.01, c_3=10.0. Initial values Y_1=Y_2=1000, X=10^5.

A more realistic system can be defined by adding a reaction that allows the prey to die of natural causes:



$$Y_1 \xrightarrow{c_4} Z$$

This can be modeled in the pi-calculus by changing the definition of the corresponding process $Y_1()$:

$$Y_1() = !c_1.(Y_1() | Y_1()) \; + \; ?c_2 \; + \; \tau_{c4}$$

The resulting system can be simulated in SPiM by taking c_4=c_3, as shown in Fig. S5. As expected, both the predator Y_2 and the prey Y_1 eventually become extinct. However, it is interesting to note that the predator species becomes extinct significantly before the prey, even though they have the same life expectancy ($1/c_3$=$1/c_4$). More surprisingly, over 40% of the initial food source remains after both the predator and prey have become extinct. These results indicate how useful (and sometimes unexpected) insight can be gained through the stochastic simulation of systems.

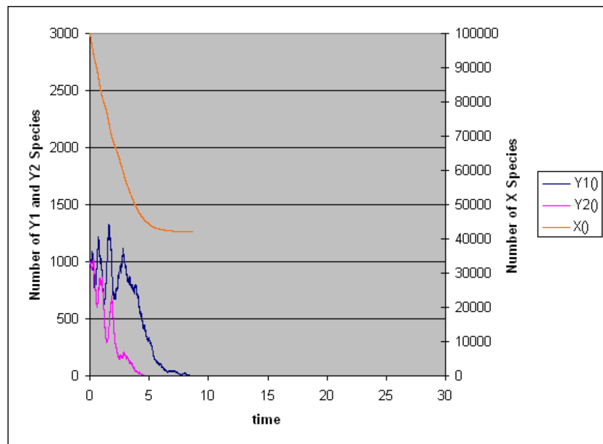


Fig. S5. Simulation results for the Lotka reactions (Eq. 6) but with limited number of X species and an additional reaction $Y_1 \rightarrow^{c_4} Z$ that allows the prey to die of natural causes. Simulation up to time t=30, with $rate(c_1)$=0.0002, $rate(c_2)$=0.01, c_3=10.0, c_4=10.0. Initial values Y_1=Y_2=1000, X=10^5.

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 Fig. 7C 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 Fig. 15,17, 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 $(!dx;\ P)\ |\ (?dx;\ Q)$, 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 $P|Q$.

$P|Q \rightarrow_r (!dx;\ P)\ |\ (?dx;\ Q) \rightarrow_s P|Q$ 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 Fig. 3 of the main paper. 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).

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. (Fig. S6 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 (Fig. S6 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 (A1,A2) and a less sensitive gate with η=1.0 (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.

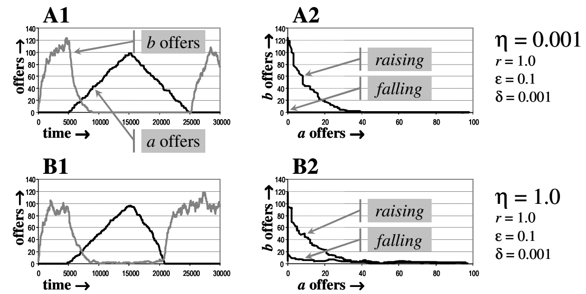


Fig. S6. Neg Gate Response Profile

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))
```