I. COMPOSITIONALITY, STOCHASTICITY AND COOPERATIVITY IN DYNAMIC MODELS OF GENE REGULATION (SUPPLEMENTARY MATERIAL)

A. A primer on the stochastic π-calculus

An ordinary *function* is a method of computing outputs from inputs; for our purposes we can say that a function has an input channel where inputs are received, and an output channel where outputs are provided. Functions are composed by ordinary *function composition*, that is, by connecting the output channel of a function to the input channel of another function, obtaining again a function. Function composition is associative: we can imagine all the composed functions lined up end-to-end, with the processing of information occurring sequentially along such a line.

A *process* is, more generally, a method that expects inputs from many channels, and produces outputs to many channels. Since there are no "default" input and output channels, the channels are explicitly named. Each channel can be used both for input and for output, and there can be many inputs separately received from one or several channels before an output is produced, or vice versa. Two processes are composed by *parallel composition*, by connecting some channels of one process to some channels of the other process (as dictated by the channel names). The two processes then proceed independently performing inputs and outputs to the environment, except when they are interacting among themselves on a shared channel, in which case the input of one process is *synchronized* to the output of the other. The parallel composition of two processes is again a process of the same kind. Parallel composition is associative and commutative, hence a collection of processes is not a line, but a "soup" where information is processed concurrently, and where each process acts as part of the environment for all the other processes.

In almost the same way that a function can invoke other functions as subroutines, a process can spawn other processes (or rather, subdivide into multiple processes). That is, a parallel composition can be enacted after an input or an output, dynamically enlarging the soup of processes. Conversely, a process can simply stop performing inputs and outputs, and hence vanish, or at least become undetectable. Channels need not be static either: new uniquely-named channels can be created dynamically to form evolving connection topologies.

π-calculus is a formal algebra; it is a remarkably compact and expressive way of capturing

the above intuition of processes, and of generalizing the standard theory of functions. It has a syntactic formulation (like any algebra), a set of algebraic laws (like commutativity of composition), but also a set of computation rules, which in fact boil down to the simple rule of matching inputs and outputs in pairs (*communication*). Like any algebra, the entire computational game can be played out over the syntax, without ever touching the models that the algebra is abstracting from. Unlike more familiar algebras, π-calculus has *variable binders* (one for inputs, and one for new channels) that restrict variables within syntactic scopes. But even that is a familiar concept from λ-calculus, which is the algebra of ordinary functional computations.

The basic concepts of π-calculus are embodied one-to-one into its syntax: creating new channels, performing an input or an output, choosing (which input or output to perform), placing two processes in composition, and dying out. In addition, some form of recursion is needed for computational completeness; this usually takes the form of *replication*, that is, producing an arbitrary number of copies of a given process. This turns out to be equivalent to simply defining and then mentioning a process recursively.

Each output on a channel, if successfully matched with an input on the same channel, communicates a value that is picked up by the corresponding input. In pure π-calculus the values that are communicated are yet other channels, as this is technically sufficient for computational expressiveness, but we can also communicate simple data of any kind. A special case is communicating *no* data (or a fixed token), which still results in the processes being synchronized with each other by the dummy communication. In *stochastic π-calculus*, each channel has a fixed assigned *rate*, which is the stochastic rate at which communications happen on that channel. Process evolution then implicitly describes a Continuous Time Markov Chain, where states are the global states of the soup of processes, and transitions are communication events that modify the state of the soup, with each transition bearing the rate of the corresponding communication channel.

Here is a simple example of a process in π-calculus, corresponding to the function x^2, where "?" indicates input and "!" indicates output. To be precise, we need to distinguish an "output" (which is an actual communication), from an "output offer", which is a pending action that may result in an actual communication, but only if successfully matched by a

corresponding "input offer". The function x^2 as a process is

$$?b(x).!c(x^2).0 \tag{1}$$

This process offers to input x on channel b, if successful then offers to output x^2 on channel c, and if successful then terminates ("0" is the dead process, not a number). A dot "." separates one state from the next.

A simple computation in π-calculus, is the following; it closely corresponds to supplying the value 7 to the function x^2, eventually obtaining 7^2.

$$\begin{array}{c} !b(7).0 \mid \\ ?b(x).!c(x^2).0 \end{array} \tag{2}$$

Here two processes are placed in parallel composition ("|"), both using the channel b. The first process offers to output 7 on channel b. The second process is the same as before, offering to input x on b and then offering to output x^2 on c. Since the two current offers can be matched, we have a communication on channel b, which results in the following two residual processes:

$$\begin{array}{c} 0 \mid \\ !c(7^2).0 \end{array} \tag{3}$$

Since $0|P$ is the same as P (this is a basic algebraic law), we are left with a process that offers to output 7^2 on channel c.

An example of a process that is not a function is instead:

$$?b(x).(!c(x^2).0+!d(x^3).0) \tag{4}$$

this process inputs (i.e., offers to input) x on channel b, then *either* outputs x^2 on channel c *or* ("+") outputs x^3 on channel d. If either of the two output choices is successful, the other choice is discarded, and in both cases the process terminates. We can similarly have choices between inputs, or mixed choices between any number of inputs and outputs.

An example of a computation that is not functional (although using a function) is the

following:

$$\begin{aligned} !b(7).0 \mid \\ !b(9).0 \mid \\ ?b(x).!c(x^2).0 \end{aligned} \tag{5}$$

Here two processes offer to output a value (7 or 9) to the x^2 function. Either can be successful, resulting in two different outcomes for the system. Suppose we match the $!b(9)$ output with the $?b(x)$ input, we then obtain the residuals:

$$\begin{aligned} !b(7).0 \mid \\ 0 \mid \\ !c(9^2).0 \end{aligned} \tag{6}$$

where the $!b(7)$ output remains unused, and 9^2 is offered over c. Note that the x^2 function has been "consumed": if we want to use it (or any process) an arbitrary number of times we can instead *replicate* it. The replication operator $*P$ is by definition algebraically equivalent to $(P \mid *P)$. Hence, if we replicate our x^2 function and we provide two outputs as above, after two expansions of replication and two communications, we are left with two output residuals on c, 7^2 and 9^2, and with the replicated function still intact:

$$\begin{aligned} !c(7^2).0 \mid \\ !c(9^2).0 \mid \\ *(?b(x).!c(x^2).0) \end{aligned} \tag{7}$$

The only remaining concept in π-calculus is the operation that creates a new channel and privately shares it among a collection of processes; "privately" means that those processes are the only ones to initially "know" that channel, although they can later communicate it to other processes. We have seen previously that an output to the x^2 function can be interfered with by a third process that operates on the same channel. This interference from the environment can be prevented by "hiding" the channel, that is, by creating a channel and sharing it only among some processes:

(νb) a new ("ν") channel b, privately shared in the following scope

$!b(7).0 \mid$ output on the private b

$?b(x).!c(x^2).0$ input on the private b, etc.

If another process "outside" (i.e., in parallel composition) with this process also happens to use a channel named b, that channel is in fact a different channel, and there is no interference with the private b. Formally, one can consistently rename a private channel, bound by ν, to any other (non internally conflicting) name, irrespectively of the outside context. In this case we could not rename b to c, because c is internally used as a public name, but we could rename b to, e.g., d, no matter whether b and d are used in the outside context. These too are basic algebraic laws of π-calculus.

In summary, a process P can be one of:

$(\nu b_\delta)P'$ create a new channel named b (with stochastic rate δ), and use it privately within process P'

$!c(b).P'$ offer to perform an output (indicated by "!"); if matched by an input on the same channel, the value b is transmitted over channel c, and then the continuation process P' is executed.

$?c(x).P'$ offer to perform an input (indicated by "?"); if matched by an output on the same channel, the value received over channel c is bound to the variable x, and then the continuation process P' is executed.

$\pi_1.P_1 + ... + \pi_n.P_n$ generalizing the previous two cases, offer to perform one (and only one) of the inputs or outputs π_i; if π_i is matched by a complementary output or input, the continuation process P_i is executed and the others are discarded.

$P_1 \mid P_2$ place processes P_1 and P_2 in parallel composition

0 terminate

$*P$ replicate P, equivalent to $(P| * P)$, i.e., an arbitrary number of P's in parallel

Basic computation rule:

$(!c(b).P_1 + ...) \mid (?c(x).P_2 + ...)$ *transitions to* $P_1 \mid (P_2\{b/x\})$

where "..." are the choices that are discarded, if any, and $P_2\{b/x\}$ is P_2 where the input variable x is bound to (substituted by) the output value b.

Other concepts we use in the paper are in fact definable within pure π-calculus, namely:

$?b.P, !b.P$ inputs and outputs with no value exchanged

$?b(x_1, ..., x_n).P$ inputs (and similarly outputs) with multiple parameters

$\tau_\delta.P$ execute process P after a stochastic delay of rate δ (defined as $(\nu c_\delta)(!c.0 \mid ?c.P)$ for any c not used in P)

$X = P$ give name X to process P, and then use it, possibly recursively. E.g., $X \mid X$, meaning $P \mid P$.

$X(y, z) = P$ give name X to process P with parameters y, z. Then, $X(b, c)$ stands for P where y and z are replaced by b and c respectively.

B. Running simulations in stochastic π-calculus

A system of chemical reactions is an implicit description of a Continuous Time Markov Chain over the state space of its solution, and the Gillespie algorithm (Gillespie D, 1977) is a correct way to simulate this Markov chain. Similarly, a stochastic π-calculus model is an alternative, implicit way to describe the same Markov chain, which is also simulated by the Gillespie algorithm.

Since the Gillespie algorithm is defined for a system of chemical reactions, a given stochastic π-calculus model must first be translated into such a system in order to be correctly simulated. A simulation algorithm for the stochastic π-calculus is described in (Phillips A and Cardelli L, 2007), which performs the translation dynamically at each reaction step, such that the set of all possible reactions of a system does not need to be computed beforehand. In the general case there can be a potentially unbounded number of reactions, since new species can be dynamically created. The algorithm can therefore handle unbounded-state systems and systems with an unbounded number of species. Full technical details of the

algorithm can be found in (Phillips A and Cardelli L, 2007), together with a proof of correctness. In this section we illustrate the algorithm by applying it to a simple auto-inhibitory gene gate based on Figure 1 of the main text, as shown in Table I.

For the chemical reaction model on the left of the figure, the Gillespie algorithm simply chooses the next reaction from the set of possible reactions, with probability proportional to the reaction rate. For the stochastic pi-calculus model on the right, the simulation algorithm needs to dynamically determine the set of possible reactions at each step. This is achieved by examining all of the running processes in the system and counting the number of inputs, outputs and delays that are executing in parallel. Each delay corresponds to a single reaction, as does each pair of parallel inputs and outputs on the same channel. Initially the system in Table I consists of a single gene, g. A possible sequence of simulation steps for stochastic the π-calculus model is described below.

1. The algorithm expands the definition of gene g and works out that there is a single delay reaction $\tau_\varepsilon.(P \mid g)$ that can be executed by the gene. There is also an input on a, but no reaction can take place on this channel since there are no corresponding outputs. The algorithm executes the delay reaction by consuming the delay τ_ε and leaving behind the process $P \mid g$.

2. The algorithm expands the definitions of gene g and protein P and works out that there are two possible reactions: a delay reaction $\tau_\varepsilon.(P \mid g)$ as before, together with a communication reaction on a between the input $?a.g'$ of the gene and the output $!a.P$ of the protein. The probability of each reaction is proportional to its rate, in accordance with the Gillespie algorithm. Hence the probability of the delay reaction is given by $\varepsilon/(\varepsilon + r)$ and the probability of the communication reaction is given by $r/(\varepsilon + r)$, where $rate(a) = r$. Note that if there were say 100 copies of protein P running in parallel with the gene $(P \mid ... \mid P \mid g)$, then the probabilities would be given by $\varepsilon/(\varepsilon + 100 \cdot r)$ and $100 \cdot r/(\varepsilon + 100 \cdot r)$, respectively. Assuming that a communication reaction on channel a is chosen, the algorithm executes this reaction by consuming an input $?a$ and output $!a$, and replacing them with $P \mid g'$.

3. The algorithm continues computing the current set of reactions and choosing the next reaction in this manner until no further reactions are possible.

In the general case the gene g can be parameterised by multiple arguments, e.g. $g(a, b)$, as shown in Figure 1 of the main text. The simulation algorithm handles this by first replacing the arguments for all of the running processes in the system, and then computing the set of reactions as described above. For example, if a process $g(b, c)$ is running then the algorithm replaces (a, b) with (b, c) in the definition of the gene, in order to compute the current set of reactions.

C. Parameter Variation for Basic Repressilator

In Section III A of the main text we used probabilistic analysis to explain why the repressilator network of Figure 1 of the main text should exhibit regular, unperturbed protein cycles for $\delta < \varepsilon/1000$, $\eta > \delta$ and $r > 100 \cdot \varepsilon \cdot \eta/\delta$. In this appendix we complement our analysis with additional simulation results. As described in Section III A of the main text, we fix the constitutive rate of protein production ε at a nominal value of 0.1, and then vary the rates of the other three parameters δ, η, r. We proceed by varying η and r for different values of protein degradation δ. We stress that these simulations are merely examples, which illustrate that our probabilistic analysis is indeed applicable.

Figure 1 shows a range of simulations for systems that satisfy the required constraints. Probabilistic analysis implies that increasing the rate of repression r will not adversely affect the system behaviour, since the higher repression rate will further prevent a given gene from producing if any residual repressors remain. As a result, most of the simulations were executed for $r = 100 \cdot \varepsilon \cdot \eta/\delta$. A range of simulations were also performed for which the required constraints were not satisfied. In these cases the protein cycles were significantly perturbed when moving beyond an order of magnitude from the given constraints (results not shown).

D. Repressilator Program Code in SPiM

This section contains the SPiM code for the Repressilator models described in the main text, as shown in Figures 2-6. Details of the SPiM Language definition are available at:

http://research.microsoft.com/~aphillip/spim/Language.pdf

	Chemical reaction model	Stochastic π-calculus model, $rate(a) = r$
	$g + A \rightarrow_r g' + A$ $g \rightarrow_\varepsilon A + g$ $g' \rightarrow_\eta g$ $A \rightarrow_\delta 0$	$g = ?a.g'$ $+ \tau_\varepsilon.(P \mid g)$ $g' = \tau_\eta.g$ $P = !a.P$ $+ \tau_\delta.0$
1	g	g
2	$A + g$	$P \mid g$
3	$A + g'$	$P \mid g'$

Table I: Initial simulation steps for an auto-inhibitory gene gate. The first row denotes the system definition in terms of chemical reactions (left) and stochastic π-calculus (right), while the remaining rows denote the evolution of the system, starting from its initial state (1).

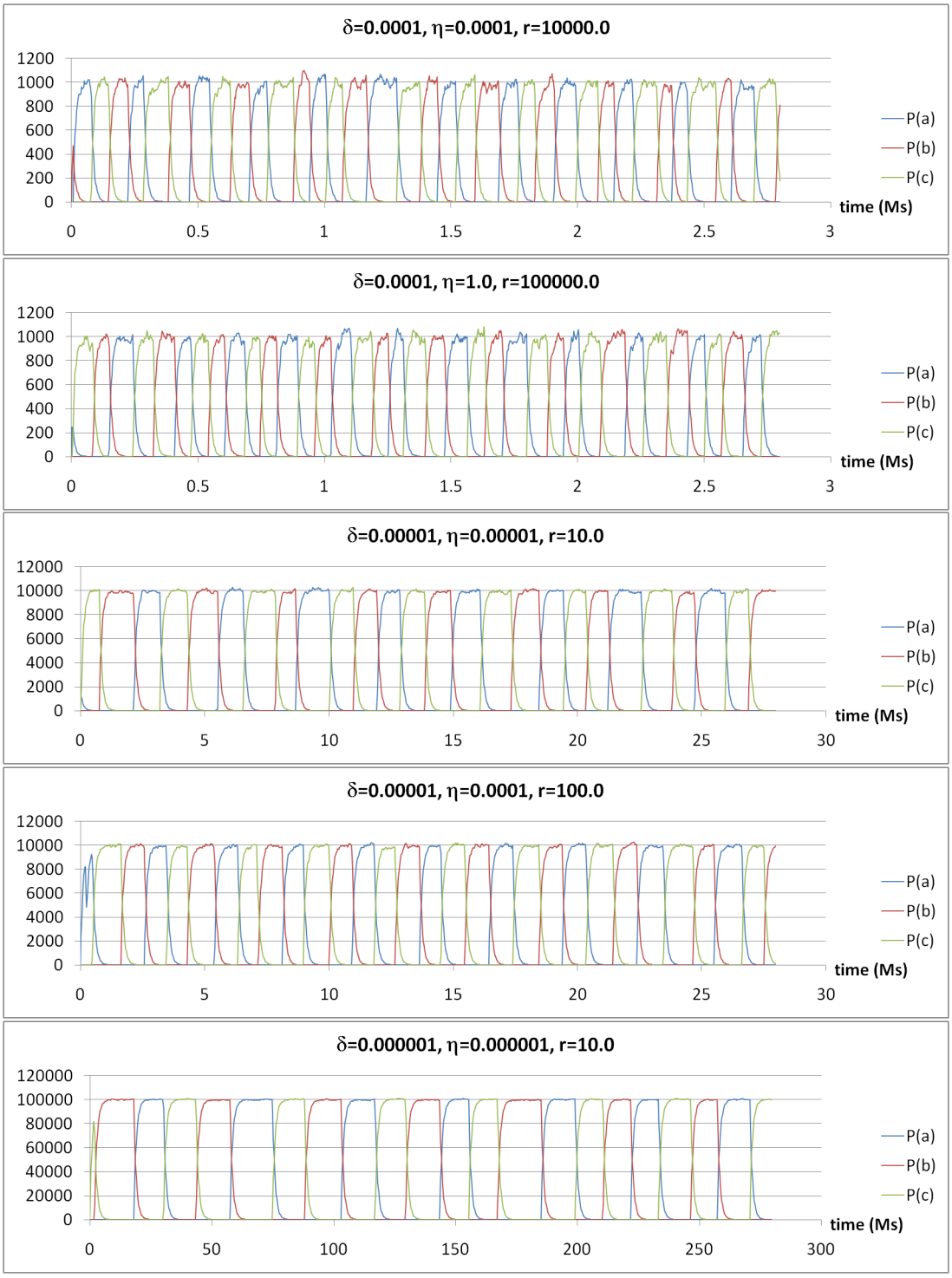

Figure 1: Simulation results for the repressilator network of Figure 1 of the main text, with $\varepsilon = 0.1$, $\delta \leq \varepsilon/1000$, $\eta \geq \delta$ and $r \geq 100 \cdot \varepsilon \cdot \eta/\delta$. The plots show the populations of proteins $P(a), P(b), P(c)$ over time (Ms). The simulations exhibit the desired behaviour of regular, unperturbed oscillations within the given parameter range.

```
(* Repressilator *)
directive sample 2000000.0 1000
directive plot !a as "P(a)"; !b as "P(b)"; !c as "P(c)"
val e = 0.1 val d = 0.0001 val h = 0.001 val r = 10.0
new a@r:chan new b@r:chan new c@r:chan
let g(a:chan,b:chan) =
  do delay@e; (p(b) | g(a,b))
  or ?a; g_(a,b)
and g_(a:chan,b:chan) = delay@h; g(a,b)
and p(b:chan) = do !b; p(b) or delay@d
run ( g(a,b) | g(b,c) | g(c,a) )
```

Figure 2: SPiM code for a basic gene gate, as described in Figure 1 of the main text.

```
(* Repressilator RNA *)
directive sample 2000000.0 1000
directive plot !a as "P(a)"; !b as "P(b)"; !c as "P(c)"
val e = 0.1 val d = 0.0001 val h = 0.0001 val r = 10.0
val e2 = 0.01 val d2 = 0.01
new a@r:chan new b@r:chan new c@r:chan
let g(a:chan,b:chan) =
  do delay@e; (m(b) | g(a,b))
  or ?a; g_(a,b)
and m(b:chan) =
  do delay@e2; (p(b) | m(b))
  or delay@d2
and g_(a:chan,b:chan) = delay@h; g(a,b)
and p(b:chan) = do !b; p(b) or delay@d
run ( g(a,b) | g(b,c) | g(c,a) )
```

Figure 3: SPiM code for a gene gate with transcription and translation, as described in Figure 4 of the main text.

```
(* Repressilator Binding *)
directive sample 2000000.0 1000
directive plot !a as "P(a)"; !b as "P(b)"; !c as "P(c)"
type bind = chan(chan)
val e = 0.1 val d = 0.0001 val h = 0.0001 val r = 10.0
new a@r:bind new b@r:bind new c@r:bind
let g(a:bind,b:bind) =
  do delay@e; (P(b) | g(a,b))
  or ?a(u); g_(a,b,u)
and g_(a:bind,b:bind,u:chan) =
(* do ?a(u_); g__(a,b,u,u_) or *) !u; g(a,b)
(* and g__(a:bind,b:bind,u:chan,u_:chan) = !u_; g_(a,b,u) *)
and P(b:bind) = (
  new u@h:chan
  do !b(u); P_(b,u)
  or delay@d
)
and P_(b:bind,u:chan) = (*do ?u or*) ?u;P(b)
run ( g(a,b) | g(b,c) | g(c,a) )
```

Figure 4: SPiM code for a gene gate with repressor binding, as described in Figure 5 of the main text. The comment (*`do ?u or*`) corresponds to degradation of bound repressors, while the remaining comments correspond to an additional repressor binding site.

```
(* Repressilator Dimers *)
directive sample 1000000.0 1000
directive plot !a as "P(a)"; !b as "P(b)"; !c as "P(c)"
val e = 0.2 val d = 0.0001 val h = 0.001
val r = 1.0 val r2 = 0.0001
new a@r:chan new b@r:chan new c@r:chan
new a2@r2:chan new b2@r2:chan new c2@r2:chan
let g(a:chan,b2:chan,b:chan) =
  do delay@e; (P(b2,b) | g(a,b2,b))
  or ?a; g_(a,b2,b)
and g_(a:chan,b2:chan,b:chan) = delay@h; g(a,b2,b)
and P(b2:chan, b:chan) = do ?b2; P2(b) or !b2; () or delay@d
and P2(b:chan) = do !b; P2(b) or delay@d
run ( g(a,b2,b) | g(b,c2,c) | g(c,a2,a) )
```

Figure 5: SPiM code for a gene gate with dimerization of repressors, as described in Figure 7(i) of the main text.

```
(* Repressilator Tetramers *)
directive sample 1000000.0 1000
directive plot !a as "P(a)"; !b as "P(b)"; !c as "P(c)"
val e = 0.4 val d = 0.0001 val h = 0.001 val r = 1.0
val r2 = 0.0001 val r4 = 0.0001
new a@r:chan new a2@r2:chan new a4@r4:chan
new b@r:chan new b2@r2:chan new b4@r4:chan
new c@r:chan new c2@r2:chan new c4@r4:chan
let g(a:chan,b4:chan,b2:chan,b:chan) =
  do delay@e; (p(b4,b2,b) | g(a,b4,b2,b))
  or ?a; g_(a,b4,b2,b)
and g_(a:chan,b4:chan,b2:chan,b:chan) = delay@h; g(a,b4,b2,b)
and p(b4:chan,b2:chan,b:chan) = do ?b2; p2(b4,b) or !b2; () or delay@d
and p2(b4:chan,b:chan) = do ?b4; p4(b) or !b4; () or delay@d
and p4(b:chan) = do !b; p4(b) or delay@d
run ( g(a,b4,b2,b) | g(b,c4,c2,c) | g(c,a4,a2,a) )
```

Figure 6: SPiM code for a gene gate with tetramerization of repressors, as described in Figure 7(ii) of the main text.