

# Towards an aspect-oriented design and modelling framework for synthetic biology - Supplementary Information

Philipp Boeing, Miriam Leon, Darren Nesbeth, Anthony Finkelstein  
and Chris P. Barnes

## Contents

<b>1</b>	<b>SynBioWeaver Implementation</b>	<b>2</b>
1.1	Overview . . . . .	2
1.2	Simple Example: RBS and Terminator Design Rules . . . . .	2
1.3	Circuitry Architecture . . . . .	3
1.3.1	Parts . . . . .	3
1.3.2	Molecules . . . . .	3
1.3.3	Dynamically creating Part and Molecule types . . . . .	3
1.3.4	Circuits . . . . .	3
1.3.5	Circuit as Composites . . . . .	4
1.4	Point Cuts . . . . .	4
1.4.1	Part Signature . . . . .	4
1.4.2	Point Cut Expressions . . . . .	5
1.5	Aspects and Advice . . . . .	5
1.5.1	Advice precedence . . . . .	6
1.6	Type Advice . . . . .	6
1.7	Weaving . . . . .	7
<b>2</b>	<b>A switchable oscillating system based on the repressilator combined with an AND gate</b>	<b>8</b>
2.1	Defining the concerns . . . . .	8
<b>3</b>	<b>Rule-based modelling as an output concern</b>	<b>11</b>
3.1	Overview . . . . .	11
3.2	Defining the Concerns . . . . .	11
<b>4</b>	<b>Generating reaction network models and adding context</b>	<b>15</b>
<b>5</b>	<b>Bayesian modelling and characterisation</b>	<b>16</b>
5.1	Code for constitutive GFP and different bacterial growth contexts . . . . .	16
5.2	Model for the core concern plus lag logistic growth . . . . .	17
<b>6</b>	<b>Post-translational coupling of a bistable switch and an oscillator</b>	<b>17</b>
6.1	Code for the combined and coupled systems . . . . .	17
6.2	Toggle switch model . . . . .	19
6.3	Stricker oscillator model . . . . .	20

# 1 SynBioWeaver Implementation

## 1.1 Overview

SynBioWeaver is a framework written in Python 2.7 that allows a synthetic biology system to be built in an aspect-oriented manner. To achieve this, SynBioWeaver provides classes to model genetic parts. Via a point cut system, the genetic circuit can be advised by influencing the execution flow. Furthermore, parts can be enriched with type advice to include additional properties, such as modelling rules. SynBioWeaver is able to combine all these advice into a finished design, which includes an ordered list of parts representing the genetic circuit. All classes are in the `synbioweaver.core` Python module.

All the code and examples are downloadable as the module `synbioweaver` from GitHub and have online documentation at the following urls:

- <https://github.com/ucl-cssb/synbioweaver>
- <http://synbioweaver.readthedocs.org>

## 1.2 Simple Example: RBS and Terminator Design Rules

To give an initial flavour for the framework, the following code listing shows a simple example. It defines a standard genetic circuit: a promoter, followed by an RBS, followed by a coding region for GFP, followed by a terminator. However, in this example the core-concern of expressing GFP under a Promoter has been separated from the cross-cutting concern of always surrounding a coding region with an RBS and a terminator.

Listing 1: SynBioWeaver Example: Design Rules

---

```
1 from synbioweaver import *
2
3 declareNewMolecule('GFP')
4
5 class CodingGFP(Circuit):
6     def mainCircuit(self):
7         self.addPart(Promoter)
8         self.addPart(CodingRegion(GFP))
9
10 class DesignRules(Aspect):
11     def mainAspect(self):
12         anyCodingRegion = PartSignature('*.CodingRegion+')
13
14         beforeCodingRegion = PointCut(anyCodingRegion, PointCut.BEFORE)
15         afterCodingRegion = PointCut(anyCodingRegion, PointCut.AFTER)
16
17         self.addAdvice(beforeCodingRegion, self.insertRBS)
18         self.addAdvice(afterCodingRegion, self.insertTerminator)
19
20     def insertRBS(self, context):
21         self.addPart(RBS)
22
23     def insertTerminator(self, context):
24         self.addPart(Terminator)
25
26 compiledDesign = Weaver(CodingGFP, DesignRules).output()
27 print compiledDesign
```

---

In the following sections, the classes used in the example will be described in detail. This includes *Part* and *Circuit* types, *point cuts* and *advice*, and the *weaver*. These relate to the two most high priority requirements:

- Circuits can be modelled as a list of parts. Parts are objects with meaningful types
- A weaver allows circuit design to be advised by design aspects

Some additional concepts are not described further, but can be found in the documentation, which also includes tutorials and examples.

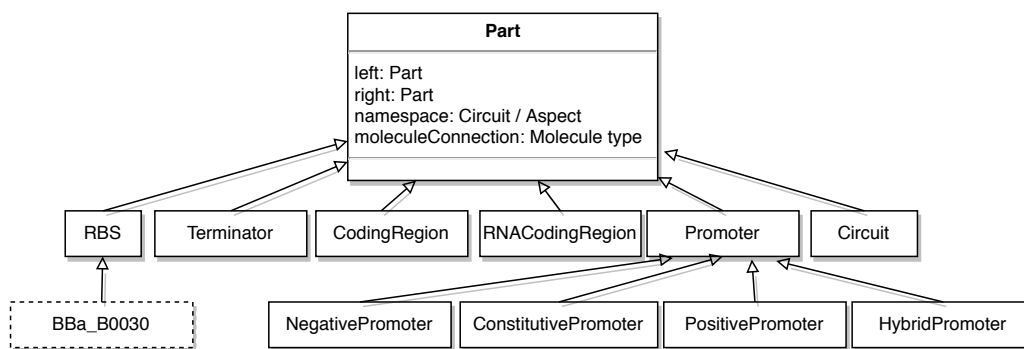


Figure S1: UML class diagram shows the built in part types in `synbioweaver.core` in solid boxes. Users can define new, more concrete part types, as shows by the BioBrick RBS part (dashed box).

## 1.3 Circuitry Architecture

### 1.3.1 Parts

Parts follow a rich inheritance hierarchy, with abstract parts of types such as “RBS” at the top, which could be subclassed into a concrete part such as “BBa\_B0030”<sup>1</sup>. Figure S1 shows all part types in `synbioweaver.core`. Parts could have been made to differ via object attributes rather than type, however, since Python allows new types to be created quickly and dynamically we believe that it is conceptually clearer to represent each distinct part type (in biology) by a distinct type in the framework.

### 1.3.2 Molecules

Some parts, such as regulated promoters or coding regions are connected to molecules. These can be proteins such as GFP, transcription factors or other chemical agents interacting with the system. Unlike parts, molecules are always only used as types in SynBioWeaver, never objects. This is because they represent a class of Molecule rather than a concrete, physical molecule. In other words, if a promoter is regulated by a certain protein, it is regulated by all proteins of the type, not just by a single instance. All molecules are of subtype `Molecule`.

### 1.3.3 Dynamically creating Part and Molecule types

Two convenience functions, `declareNewPart()` and `declareNewMolecule()` are provided by SynBioWeaver. As seen in the example, they export the string name of the new type to the namespace. More information can be found in the documentation.

Listing 2: SynBioWeaver example of new Part and Molecule type declaration

```

1 # Protein is a subtype of Molecule
2 declareNewMolecule('GFP',Protein)
3
4 # GFP is now a new type in this namespace
5 declareNewPart('GFPcoding',CodingSite,GFP)
6
7 # a new part type has been created, getCodingFor returns a list of coding molecules
8 assert(GFPcoding().getCodingFor()[0] == GFP)
  
```

### 1.3.4 Circuits

A circuit is an ordered collection of parts. Circuits are given an `addPart()` method, and a mandatory `mainCircuit()` method, at which the circuit’s execution begins. This design decision makes the circuit more

<sup>1</sup>A BioBrick part from the Registry of Standard Biological Parts, in the RBS category. [http://parts.igem.org/Part:BBa\\_B0030](http://parts.igem.org/Part:BBa_B0030)

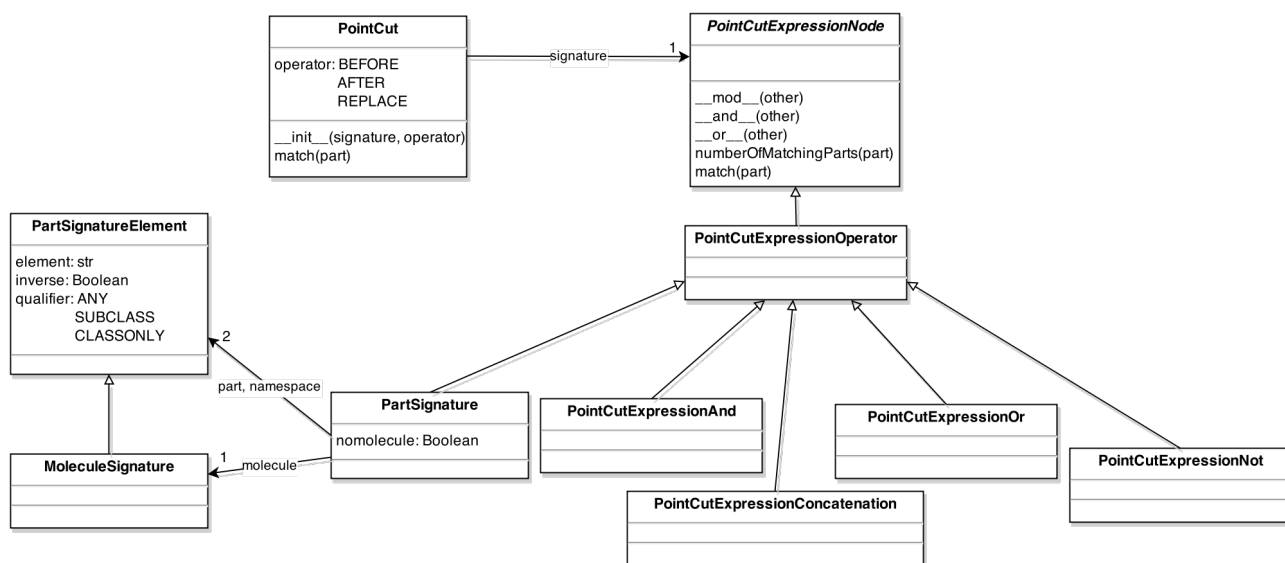


Figure S2: UML class diagram of classes involved in the Point Cut mechanism

like a dynamic programmatic construct, rather than a specified list. Indeed, all normal Python code can be used, such as loops and method calls to construct the genetic circuit with `addPart()` calls. The `Circuit` class is instantiated by the weaver, which then calls its `mainCircuit()` method. Whenever the circuit calls `addPart()`, the method is passed on back to the weaver, which decides if any point cuts are relevant. Thus each call to `addPart()` is an SynBioWeaver join point.

### 1.3.5 Circuit as Composites

As seen in the `Part` inheritance diagram (Figure S1), circuits are themselves parts. This means they can be used as composites, which reflects the idea of composite parts from the Registry of Standard Biological Parts. If a circuit is used as a composite, its `mainCircuit()` method is called when it is added via `addPart()`.

## 1.4 Point Cuts

A point cut in SynBioWeaver has to be able to select join points in the genetic circuit execution flow, i.e. calls to `addPart()`. Inspired by AspectJ, point cuts should be able to precisely select a single join point, or be able to select whole groups of join points based on features such as type name or type hierarchy. A SynBioWeaver `PointCut` object consists of an expression to select certain parts - the parameter to the `addPart()` call, and an operator to declare whether the advice should be inserted before or after the part is added, or whether the advice should replace the part. All classes involved in the point cut mechanism are shown in Figure S2.

### 1.4.1 Part Signature

A part signature is a single string that can match a part or group of parts. It consists of three elements. Firstly, it can specify a *namespace* of a `Part`, i.e. the circuit or aspect which has added it. Secondly, it can specify the part type. Thirdly, it can optionally specify a molecule the part should be connected to. Elements are matched by typename. A `+` at the end of each element specifies that all subclasses should be matched, `*` at the end of an element acts as a string wildcard, i.e. all types whose names begin with the characters preceding the wildcard will be matched. Each element can be inverted using a `!` at the beginning. 1 shows examples of valid and invalid signatures. Strings are validated and split using regular expressions, and the overall `PartSignature` object is implemented using classes for each element, so that matching a signature becomes the conjunction of matching the three elements.

Part Signatures	Matched Parts
SimpleCircuit.BBa_B0030	Parts added in SimpleCircuit of type BBa_B0030.
Simple*.NegativePromoter+(TeTR)	Parts of type NegativePromoter (including subtypes) added in Aspects or Circuits whose typename begins with “Simple”, which have a moleculeConnection of type TeTR
!SimpleAspect.*(Protein+)	Any Part with a connection to a Protein added by an Aspect or Circuit whose name is not “SimpleAspect”
*.*(!TetR)	Any Part from any Circuit or Aspect whose moleculeConnection is not TeTR
*.Promoter+()	Any Part of (sub)type Promoter, whose moleculeConnection is None.

Table 1: Examples of Part Signatures

### 1.4.2 Point Cut Expressions

Often, a single part signature will be sufficient to define a point cut, but sometimes more control is needed. SynBioWeaver allows part signatures to be building blocks of more complex expressions, which represent an Abstract Syntax Tree (AST) of part signatures. Part signatures can be combined using boolean-or and boolean-and. A signature can also be concatenated to another signature, so that we can define join points based on a number of preceding parts. Lastly, a signature can be inverted.

`PointCutExpressionOr`, `PointCutExpressionAnd` and `PointCutExpressionConcatenate` nodes can be created using overloaded operators `|`, `&` and `%`, respectively. Inversion has no overloaded operator, but is created using a `PointCutExpressionNot` node. An example of a point cut expression and its corresponding AST can be seen in Figure S3.

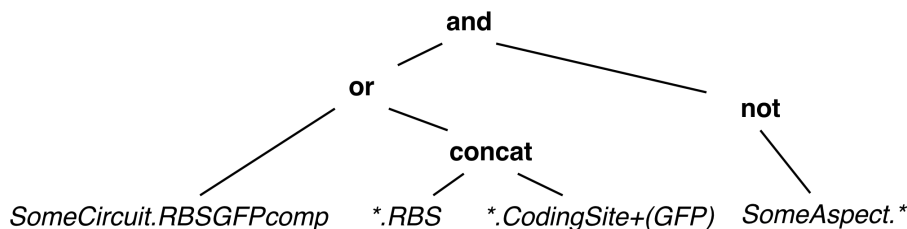


Figure S3: An AST of a point cut expression: `((PartSelector('SomeCircuit.RBSGFPcomp') | (PartSelector('*.RBS') % PartSelector('*.CodingSite+(GFP)')) & PointCutExpressionNot(PartSelector('SomeAspect.*')))`

## 1.5 Aspects and Advice

Advice is the code that should be inserted at join points matched by point cuts. For SynBioWeaver, this means that advice should also be able to use `addPart()`. An advice is bundled into an *aspect*. Each aspect is a subclass of the abstract `Aspect` class, and must implement a `mainAspect()` method, in which all advice is declared, using `addAdvice()`. This takes three parameters: a point cut, a method of the aspect that is to be called at the appropriate join point, and an optional precedence parameter. Advice methods belong to the corresponding aspect, and thus can access aspect attributes via `self`. They also receive a `PointCutContext` parameter that can access the matched part and “advice stack” information.

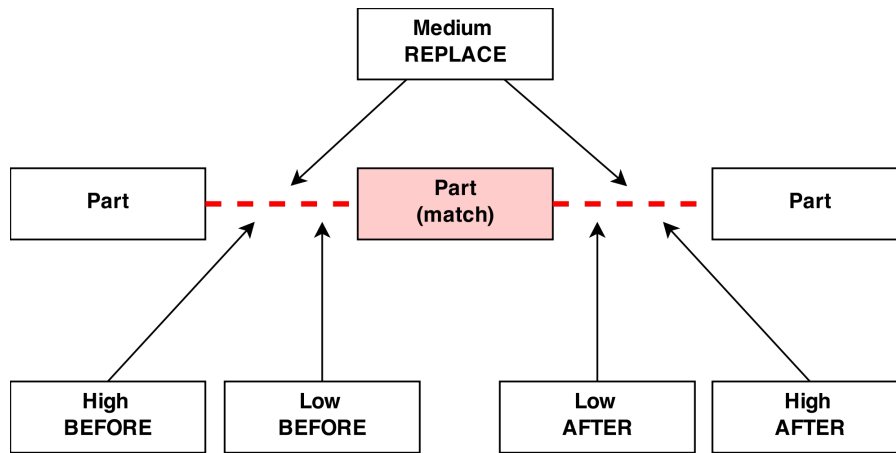


Figure S4: Five point cut / advice pairs of different operators and precedence match the same part. The low precedence *before* and *after* advice will not be executed, because of the medium precedence *replace* advice.

### 1.5.1 Advice precedence

If more than one point cut matches a join point, order of advice execution is undefined unless precedence is explicitly set via the third parameter to `addAdvice()`. `precedence` is an integer between `Advice.MINPRECEDENCE` and `Advice.MAXPRECEDENCE`. By default, every advice is set to minimal precedence. If two advices have the same precedence, execution order is also undefined. For two advices sharing a *before* point cut, the higher precedence advice will execute **before** the lower precedence advice. Thus, if both advices add a part, the part from the high precedence advice will be first in the resulting circuit. Similarly, for two advices sharing an *after* point cut, the higher precedence advice will execute **after** the low precedence advice. These precedence rules are similar to precedence in AspectJ. For two advices sharing a *replace* point cut, the higher precedence advice will execute, whereas all lower advices will be ignored. This includes lower precedence *before* and *after* advices. However, *after* advices with similar or higher precedence will still be executed. Figure S4 exemplifies the precedence rules.

## 1.6 Type Advice

Apart from normal advice, which changes the execution flow, it is also possible to modify the `Molecule` and `Part` types using type advice. For example, one aspect could add sequence information to a whole range of parts; another could add modelling rules to parts based on features selected by a `PartSignature`. As a demonstration, below is a simple example to add a method to Promoter types to print if they are regulated or not.

Listing 3: A simple example of using type advice

```

1 from synbioweaver import *
2
3
4 class TwoPromoters(Circuit):
5     def mainCircuit(self):
6         self.addPart(Promoter)
7         self.createMolecule(Protein)
8         self.addPart(NegativePromoter(Protein))
9
10 class PromoterTypeAdvice(Aspect):
11     def mainAspect(self):
12         # define three signatures
13         nonRegulatedPromoterSignature = PartSignature('*.Promoter+()')
14         regulatedPromoterSignature = PartSignature('*.Promoter+(Molecule+)')
```

```

15     anyPromoterSignature = PartSignature('*.Promoter+')
16
17     # add Boolean Attribute "isRegulated" to two different types of Promoters
18     self.addTypeAdvice(nonRegulatedPromoterSignature,False,"isRegulated")
19     self.addTypeAdvice(regulatedPromoterSignature,True,"isRegulated")
20     self.addTypeAdvice(MoleculeSignature('*..*'),4,"valueNumber")
21     self.addTypeAdvice(MoleculeSignature('*..*'),self.printNumber,"printValue")
22
23     # add a type advice method
24     self.addTypeAdvice(anyPromoterSignature,self.printIsRegulated,"printIsRegulated")
25
26     def printNumber(self,mol):
27         print "molecule value is "+str(mol.valueNumber)
28
29     def printIsRegulated(self,part):
30         if part.isRegulated == False:
31             print str(part)+" is not regulated by a Molecule."
32         else:
33             print str(part)+" is regulated by a Molecule."
34
35     compiledDesign = Weaver(TwoPromoters,PromoterTypeAdvice).output()
36     for part in compiledDesign.partList:
37         part.printIsRegulated()
38
39     for molecule in compiledDesign.moleculeList:
40         molecule.printValue()

```

---

This give the output:

Promoter is not regulated by a Molecule.

NegativePromoter(regulatedBy = Protein) is regulated by a Molecule.

molecule value is 4

## 1.7 Weaving

The **Weaver** object represents the SynBioWeaver “compiler”, taking a single **Circuit** class (which can use other circuits as composites) and a number of **Aspect** classes. Advice from aspects is automatically woven into the part based execution flow, at join points matched by point cuts, resulting in a single sorted list of parts. Figure S5 shows a sequence diagram demonstrating weaving a single before advice into a circuit.

Schematically, this is how the Weaver works:

- Call each aspect’s **mainAdvice()** method and construct a list of advice, including type advice and advice defining additional output targets for the weaver.
- Sort before, replace and after advice lists according to precedence.
- Begin circuit construction by calling the **Circuit** **mainCircuit()** method.
- At some point, **mainCircuit()** will call the Weaver’s **addPart()** method. At this point, advice might be inserted, which itself may call **addPart()**. Thus, this process is recursive. **addPart()** proceeds as follows:
  - Initialise the part and add any type advice.
  - If any before or replace advice have matching point cuts, run these.
  - Unless replaced, append the part to the list of parts.
  - Run any remaining after advice that have matching point cuts and appropriate precedence.
- After the outermost **addPart()** call and **mainCircuit()** has returned, create a **WeaverOutput** object to hold the results, including the list of parts and a list of involved molecules.





```

2 declareNewMolecule('exIn')
3 declareNewMolecule('GFP')
4 declareNewPart('cGFP', CodingRegion, moleculesAfter=[GFP] )
5
6 class CodingGFP(Circuit):
7     # constitutively code gfp
8     def mainCircuit(self):
9         declareNewPart('Pconst', ConstitutivePromoter)
10        self.addPart(Pconst)
11        self.addPart(cGFP)
12
13 class InduceGFP(Circuit):
14     # code GFP dependent on an external inducer
15
16     def getSignal(self):
17         return exIn
18
19     def getOutputName():
20         return cGFP
21
22     def mainCircuit(self):
23         self.createMolecule(exIn)
24         declareNewPart('Pin', PositivePromoter, [exIn] )
25         self.addPart(Pin)
26         self.addPart(cGFP)

```

Next, we define a composite circuit that can create oscillating behaviour. It consists of three promoter and coding region pairs, where each coded protein represses the promoter for the next coding region<sup>2</sup>.

Listing 5: Repressilator Example: Abstract Repressilator Circuit

```

1 class AbstractRepressilatorComposite(Circuit):
2     def getSignal(self):
3         return A
4
5     def mainCircuit(self):
6         declareNewMolecule('A')
7         declareNewMolecule('B')
8         declareNewMolecule('C')
9         self.createMolecule(A)
10        self.createMolecule(B)
11        self.createMolecule(C)
12
13        declareNewPart('p1', NegativePromoter, [C] )
14        declareNewPart('p2', NegativePromoter, [A] )
15        declareNewPart('p3', NegativePromoter, [B] )
16        declareNewPart('cA', CodingRegion, moleculesAfter=[A] )
17        declareNewPart('cB', CodingRegion, moleculesAfter=[B] )
18        declareNewPart('cC', CodingRegion, moleculesAfter=[C] )
19        self.addPart( p1 )
20        self.addPart( cA )
21        self.addPart( p2 )
22        self.addPart( cB )
23        self.addPart( p3 )

```

<sup>2</sup>The design is based on Elowitz and Leibler (2000), with transcription factors made abstract. They will later be re-instantiated using another aspect.

24 self.addPart( cC )

Now, we design an aspect that can weave oscillation into either of the two coding circuits. This means, the aspect needs to be able to change the promoter of the coding circuit to make coding dependent on oscillation. If the promoter is not regulated, it must be replaced by one that is regulated by oscillation output. However, if the promoter is already regulated, we need to insert an additional And-Gate to preserve the original input<sup>3</sup>.

Listing 6: Repressilator Example: Repressilation aspect

```
1 class Repressilation(Aspect):
2     # weave repressilator into a one-protein coding circuit
3     def mainAspect(self):
4         self.repressilatorComposite = AbstractRepressilatorComposite()
5
6         # promoter is unregulated (not from this aspect or the composite)
7         unregulatedPromoterReplace = PointCut(
8             PartSignature('!AbstractRepressilatorComposite.Promoter+()') &
9             PartSignature('!Repressilation.Promoter+()'), PointCut.REPLACE)
10        self.addAdvice(unregulatedPromoterReplace, self.insertRegulatedBySignalPromoter,
11                        10)
12
13        # regulated case
14        regulatedCodingExpressionReplace = PointCut(
15            (PartSignature('InduceGFP.PositivePromoter+') &
16             PartSignature('!AbstractRepressilatorComposite.Promoter+(Molecule+)') &
17             PartSignature('!Aspect+.Promoter+(Molecule+)') &
18             PartSignature('!AndGate.Promoter+(Molecule+)')) % PartSignature('*.
19                CodingRegion+(Molecule+)'),
20            PointCut.REPLACE)
21        self.addAdvice(regulatedCodingExpressionReplace, self.insertBooleanAndSignal, 10)
22
23        # add repressilator composite
24        promoterAndCodingRegion = PartSignature('Repressilation.Promoter+') %
25            PartSignature('*.RBS+') % PartSignature('*.CodingRegion+') % PartSignature('
26                *.Terminator+')
27        andGateRegion = PartSignature('Repressilation.AndGate')
28        afterCircuit = PointCut(andGateRegion | promoterAndCodingRegion, PointCut.AFTER)
29        self.addAdvice(afterCircuit, self.insertComposite, 10)
30
31        def insertRegulatedBySignalPromoter(self, context):
32            signal = self.repressilatorComposite.getSignal()
33            context.within[0].createMolecule( signal )
34            declareNewPart('ppA', PositivePromoter, [signal] )
35            self.addPart(ppA)
36            return True
37
38        def insertBooleanAndSignal(self, context):
39            originalpromoter = context.part.before[0]
40            originalsignal = context.part.before[0].getRegulatedBy()[0]
41            originaloutput = context.part.getCodingFor()[0]
42            andGate = AndGate()
43
44            # Creat first input for AND gate
45            context.within[0].createMolecule( exIn )
```

<sup>3</sup>The code of the And-Gate Circuit is not reproduced here but can be found in the complete code of the example. The design is based on Wang et al. (2011).

```

42     andGate.moleculeInputOne = exIn
43
44     if isinstance(originalpromoter, PositivePromoter):
45         andGate.setInputOneRepressing = False
46     else:
47         andGate.setInputOneRepressing = True
48
49     # Create second input for AND gate
50     in2 = self.repressilatorComposite.getSignal()
51     context.within[0].createMolecule( in2 )
52     andGate.moleculeInputTwo = in2
53
54     andGate.setInputOneRepressing = True
55     andGate.moleculeOutput = originaloutput
56
57     self.addPart(andGate)
58     return True
59
60 def insertComposite(self, context):
61     context.within[0].createMolecule( A )
62     context.within[0].createMolecule( B )
63     context.within[0].createMolecule( C )
64     self.addPart(self.repressilatorComposite)
65     return True

```

---

## 3 Rule-based modelling as an output concern

### 3.1 Overview

To demonstrate how a rule-based model could be generated as an output concern from a genetic circuit, we implemented an aspect that annotates the genetic parts in the system with appropriate reaction rules, which can then be used in a stochastic simulation or to generate ODEs. The reaction rules were based on a Kappa model of the Repressilator by Stewart and Wilson-Kanamori (2011) and include:

- Transcription Factor Binding and Un-Binding to Promoter parts
- RNAP binding to Promoter parts based on various configurations of transcription factors.
- RNAP Progression (Transcription) from Promoter parts downstream until Terminator parts resulting in an mRNA part chain
- Ribosome binding to mRNA RBS part
- Ribosome translation of the Protein Coding Region
- Protein Degradation
- mRNA Degradation

### 3.2 Defining the Concerns

The Repressilator was encoded in the circuit with parts named after their appropriate counterparts in the Parts Registry.

Listing 7: Extract from the PYSBmodel aspect

---

```

1  # transcription factor used in the repressilator system
2  declareNewMolecule('LacI')
3  declareNewMolecule('TetR')
4  declareNewMolecule('lCl')
5
6  declareNewPart('BBaB0011', Terminator)
7  declareNewPart('BBaB0034', RBS)
8  declareNewPart('BBaC0012', CodingRegion, moleculesAfter=[LacI])
9  declareNewPart('BBaC0040', CodingRegion, moleculesAfter=[TetR])
10 declareNewPart('BBaC0051', CodingRegion, moleculesAfter=[lCl])
11 declareNewPart('BBaR0010', NegativePromoter, moleculesBefore=[LacI])
12 declareNewPart('BBaR0040', NegativePromoter, moleculesBefore=[TetR])
13 declareNewPart('BBaR0051', NegativePromoter, moleculesBefore=[lCl])
14
15
16 declareNewMolecule('GFP')
17
18 class Repressilator(Circuit):
19     def mainCircuit(self):
20         self.createMolecule(LacI)
21         self.addPart(BBaR0010)
22         self.addPart(BBaC0040)
23         self.addPart(BBaR0040)
24         self.addPart(BBaC0051)
25         self.addPart(BBaR0051)
26         self.addPart(BBaC0012)

```

---

The rule-based modelling concern is implemented using a parameterized aspect, so that all of the parameters used in the model can be easily customized. The default values used in the example follow the model by Stewart and Wilson-Kanamori (2011).

Listing 8: Repressilator Circuit Specification

---

```

1  class PYSBmodel(Aspect):
2      observableProteins = []
3
4      def __init__(self, rnapCount = 700, ribosomeCount = 18000, operonCount = 1,
5                  rnapBindingHigh = 0.0007, rnapBindingLow=7e-07,
6                  transcriptionRate = 10, transcriptionFactorBinding = 0.01,
7                  transcriptionFactorUnbinding = 0.09, transcriptionFactorUnbindingSole
8                      =2.24,
9                  transcriptionFactorFactorDegradation=0.00115, rbsBinding = 0.000166,
10                 translationInitiationRate=0.167, translationRate=10, ribosomeFalloff =
11                     0.01,
12                 rnapFalloffRate=1, rnaDegradation=0.0058):

```

---

The rules and model are implemented in PySB, a python based rule-based modelling language (Lopez et al., 2013). When the aspect is initialized, it creates a PySB model and annotates each part in the system with specific type advice. For example, a PySB monomer for a part's DNA piece and potential RNA piece are added to the model. Additionally, the necessary reaction rules are added to the model, according to the parts in the system. A Terminator's rule is relatively straightforward:

Listing 9: PYSB rule for Transcription Termination

---

```

1      pysb.Rule('Transcription_Termination_' + part.pysbname,
2              self.DNA(binding = 1, partType = part.pysbname) +

```

---

```

3         self.RNAP(dna = 1,rna = 2) +
4         self.RNA(downstream = 2)
5         >>
6         self.DNA(binding = None, partType = part.pysbname) +
7         self.RNAP(dna = None,rna = None) +
8         self.RNA(downstream = None),
9         self.TranscriptionRate)

```

On the other hand, the whole method to add rules for a Negative Promoter with two binding sites is quite long:

Listing 10: PYSB rule for Transcription Termination

```

1  def pysbPartRulesNegPromoter(self,part):
2      regulatorMol = part.getBeforeNodes(Molecule)[0]
3
4      # transcription factor binding
5      pysb.Rule('TF_Binding_to_' + part.pysbname + 'p2' + '_noTF',
6              self.DNA(binding = None,partType = part.pysbname + 'p3',upstream = 2) +
7              regulatorMol.pysbmonomer(dna = None) +
8              self.DNA(downstream = 2,binding = None,partType = part.pysbname + 'p2')
9              <>
10             self.DNA(binding = None,partType = part.pysbname + 'p3',upstream = 3) +
11             regulatorMol.pysbmonomer(dna = 1) +
12             self.DNA(downstream = 3,binding = 1,partType = part.pysbname + 'p2'),
13             self.TranscriptionFactorBinding, self.TranscriptionFactorUnbindingSole)
14
15     pysb.Rule('TF_Binding_to_' + part.pysbname + 'p2' + '_TF',
16             self.DNA(binding = 1,partType = part.pysbname + 'p3',upstream = 2) +
17             regulatorMol.pysbmonomer(dna = 1) +
18             self.DNA(downstream = 2,binding = None,partType = part.pysbname + 'p2') +
19             regulatorMol.pysbmonomer(dna = None)
20             <>
21             self.DNA(binding = 2,partType = part.pysbname + 'p3',upstream = 3) +
22             regulatorMol.pysbmonomer(dna = 2) +
23             self.DNA(downstream = 3,binding = 1,partType = part.pysbname + 'p2') +
24             regulatorMol.pysbmonomer(dna = 1),
25             self.TranscriptionFactorBinding,self.TranscriptionFactorUnbinding)
26
27     pysb.Rule('TF_Binding_to_' + part.pysbname + 'p3' + '_noTF',
28             self.DNA(binding = None,partType = part.pysbname + 'p3',upstream = 2) +
29             regulatorMol.pysbmonomer(dna = None) +
30             self.DNA(downstream = 2,binding = None,partType = part.pysbname + 'p2')
31             <>
32             self.DNA(binding = 1,partType = part.pysbname + 'p3',upstream = 3) +
33             regulatorMol.pysbmonomer(dna = 1) +
34             self.DNA(downstream = 3,binding = None,partType = part.pysbname + 'p2'),
35             self.TranscriptionFactorBinding,self.TranscriptionFactorUnbindingSole)
36
37     pysb.Rule('TF_Binding_to_' + part.pysbname + 'p3' + '_TF',
38             self.DNA(binding = None,partType = part.pysbname + 'p3',upstream = 2) +
39             regulatorMol.pysbmonomer(dna = 1) +
40             self.DNA(downstream = 2,binding = 1,partType = part.pysbname + 'p2') +
41             regulatorMol.pysbmonomer(dna = None)
42             <>
43             self.DNA(binding = 1,partType = part.pysbname + 'p3',upstream = 3) +
44             regulatorMol.pysbmonomer(dna = 2) +

```

```

45         self.DNA(downstream = 3, binding = 2, partType = part.pysbname + 'p2') +
46         regulatorMol.pysbmonomer(dna = 1),
47         self.TranscriptionFactorBinding, self.TranscriptionFactorUnbinding)
48
49     # RNAP binding
50     pysb.Rule('RNAP_Binding_To_' + part.pysbname + '_no_TF',
51             self.DNA(binding = None, partType = part.pysbname + 'p3', upstream = 2,
52                     downstream = 1) +
53             self.DNA(binding = None, partType = part.pysbname + 'p4', upstream = 1) +
54             self.RNAP(dna = None, rna = None) +
55             self.DNA(binding = None, partType = part.pysbname + 'p2', downstream = 2)
56             >>
57             self.DNA(binding = None, partType = part.pysbname + 'p3', upstream = 3,
58                     downstream = 1) +
59             self.DNA(binding = 2, partType = part.pysbname + 'p4', upstream = 1) +
60             self.RNAP(dna = 2, rna = None) +
61             self.DNA(binding = None, partType = part.pysbname + 'p2', downstream = 3),
62             self.RNAPBindingHigh)
63
64     pysb.Rule('RNAP_Binding_To_' + part.pysbname + '_TF_on_p2',
65             self.DNA(binding = None, partType = part.pysbname + 'p3', upstream = 2,
66                     downstream = 1) +
67             self.DNA(binding = None, partType = part.pysbname + 'p4', upstream = 1) +
68             self.RNAP(dna = None, rna = None) +
69             self.DNA(binding = 3, partType = part.pysbname + 'p2', downstream = 2) +
70             regulatorMol.pysbmonomer(dna = 3)
71             >>
72             self.DNA(binding = None, partType = part.pysbname + 'p3', upstream = 3,
73                     downstream = 1) +
74             self.DNA(binding = 2, partType = part.pysbname + 'p4', upstream = 1) +
75             self.RNAP(dna = 2, rna = None) +
76             self.DNA(binding = 4, partType = part.pysbname + 'p2', downstream = 3) +
77             regulatorMol.pysbmonomer(dna = 4),
78             self.RNAPBindingLow)
79
80     pysb.Rule('RNAP_Binding_To_' + part.pysbname + '_TF_on_p3',
81             self.DNA(binding = 3, partType = part.pysbname + 'p3', upstream = 2,
82                     downstream = 1) +
83             self.DNA(binding = None, partType = part.pysbname + 'p4', upstream = 1) +
84             self.RNAP(dna = None, rna = None) +
85             self.DNA(binding = None, partType = part.pysbname + 'p2', downstream = 2) +
86             regulatorMol.pysbmonomer(dna = 3)
87             >>
88             self.DNA(binding = 4, partType = part.pysbname + 'p3', upstream = 3,
89                     downstream = 1) +
90             self.DNA(binding = 2, partType = part.pysbname + 'p4', upstream = 1) +
91             self.RNAP(dna = 2, rna = None) +
92             self.DNA(binding = None, partType = part.pysbname + 'p2', downstream = 3) +
93             regulatorMol.pysbmonomer(dna = 4),
94             self.RNAPBindingLow)
95
96     pysb.Rule('RNAP_Binding_To_' + part.pysbname + '_TF_on_p2_and_p3',
97             self.DNA(binding = 3, partType = part.pysbname + 'p3', upstream = 2,
98                     downstream = 1) +
99             self.DNA(binding = None, partType = part.pysbname + 'p4', upstream = 1) +

```

```

93         self.RNAP(dna = None,rna = None) +
94         self.DNA(binding = 4,partType = part.pysbname + 'p2',downstream = 2) +
95         regulatorMol.pysbmonomer(dna = 3) +
96         regulatorMol.pysbmonomer(dna = 4)
97     >>
98     self.DNA(binding = 4,partType = part.pysbname + 'p3',upstream = 3,
99             downstream = 1) +
100     self.DNA(binding = 2,partType = part.pysbname + 'p4',upstream = 1) +
101     self.RNAP(dna = 2,rna = None) +
102     self.DNA(binding = 5,partType = part.pysbname + 'p2',downstream = 3) +
103     regulatorMol.pysbmonomer(dna = 4) +
104     regulatorMol.pysbmonomer(dna = 5),
105     self.RNAPBindingLow)
106
107     # Transcription
108     pysb.Rule('Transcription_Initiation_Of_' + part.pysbname,
109             self.DNA(binding = 1,partType = part.pysbname + 'p4',downstream = 2) +
110             self.RNAP(dna = 1,rna = None) +
111             self.DNA(upstream = 2,binding = None)
112             >>
113             self.DNA(binding = None,partType = part.pysbname + 'p4',downstream = 3) +
114             self.RNAP(dna = 1,rna = 2) +
115             self.DNA(upstream = 3,binding = 1) +
116             self.RNA(binding = None,upstream = None,downstream = 2,partType = part.
117                     pysbname),
118             self.TranscriptionRate)

```

PySB was chosen because of its availability for Python and because it can create ODEs from the model or act as a translator to other established rule-based modelling languages such as Kappa or BNG.

## 4 Generating reaction network models and adding context

To demonstrate weaving at the model level, we have implemented a number of aspects for reaction network generation and contextual cross-cutting concerns. We chose to separate model generation from contextual concerns to enable consistency when called from the `WeaverOutput` object.

The current model generation aspects are as follows:

- **PromoterMapping** - this is required to map coding regions to promoters
- **MassActionKineticsProtein** - assumes mass action kinetics including only protein level abstraction
- **MassActionKineticsRNA** - assumes mass action kinetics including both protein and mRNA level abstraction
- **SheaAckersKineticsRNA** - assumes Shea-Ackers equilibrium dynamics at the promoters and mass action kinetics otherwise, including both protein and mRNA level abstraction

These aspects add the method `getReactions` to the `WeaverOutput` object. All downstream aspects that require the reactions generated must check that this method exists using `getattr(weaverOutput, "getReactions")`. On the other hand, design contextual aspects call `getReactions` but add the method `getContext`. Examples of these are

- **LagLogisticGrowth**
- **PostTranslationalCoupling**

which are described in the next two sections. Downstream analysis aspects should always use the `getContext` method over the `getReactions` method to ensure that the most up-to-date model is used. Examples of these are

- `PrintReactionNetwork`
- `WriteSBMLModel`

This is one possible choice of constraining how aspects interact at the model level and there may be other ways. One restriction of this approach is that only one model generation aspect, and one contextual aspect can be called. In future developments of the framework it would be advantageous to allow multiple reaction and contextual aspects.

## 5 Bayesian modelling and characterisation

### 5.1 Code for constitutive GFP and different bacterial growth contexts

In this example a simple model for the constitutive expression of GFP (core concern) is combined with the environmental context (cross-cutting concern) of bacterial growth. This example also shows how SBML models can be generated under different contexts.

Listing 11: Adding a bacterial growth cross-cutting concern

---

```

1 from synbioweaver.aspects.writeCudaFileODE import *
2 from synbioweaver.aspects.writeABCInputFileODE import *
3 from synbioweaver.aspects.writeCudaFileGillespie import *
4 from synbioweaver.aspects.writeABCInputFileGillespie import *
5
6 declareNewMolecule('GFP')
7
8 class constGFP(Circuit):
9     def mainCircuit(self):
10         # constitutive promoter
11         self.addPart(ConstitutivePromoter)
12         self.addPart(CodingRegion(GFP))
13
14 print "##### circuit + exp growth"
15 compiledDesign1 = Weaver(constGFP, DesignRules, PromoterMapping,
16                           MassActionKineticsProtein, ExpGrowth, PrintReactionNetwork, WriteSBMLModel).output()
17 compiledDesign1.printReactionNetwork()
18 modelStr = compiledDesign1.writeSBMLModel()
19
20 # write an SBML model out
21 sbmlFile = open("constGFP_exp.sbml", "w")
22 print >>sbmlFile, modelStr
23 sbmlFile.close()
24
25 print "##### circuit + logistic growth"
26 compiledDesign2 = Weaver(constGFP, DesignRules, PromoterMapping,
27                           MassActionKineticsProtein, LogisticGrowth, PrintReactionNetwork, WriteSBMLModel).
28                           output()
29 compiledDesign2.printReactionNetwork()
30 modelStr = compiledDesign2.writeSBMLModel()
31
32 # write an SBML model out
33 sbmlFile = open("constGFP_log.sbml", "w")

```



```

31 print >>sbmlFile, modelStr
32 sbmlFile.close()
33
34 print "##### circuit + lag logistic growth"
35 compiledDesign3 = Weaver(constGFP, DesignRules, PromoterMapping,
    MassActionKineticsProtein, LagLogisticGrowth, PrintReactionNetwork, WriteSBMLModel).
    output()
36 compiledDesign3.printReactionNetwork()
37 modelStr = compiledDesign3.writeSBMLModel()
38
39 # write an SBML model out
40 sbmlFile = open("constGFP_lag.sbml", "w")
41 print >>sbmlFile, modelStr
42 sbmlFile.close()

```

---

## 5.2 Model for the core concern plus lag logistic growth

The model for the core concern is a basic birth death system generated using the `MassActionKineticsProtein` aspect.



The contextual aspect `LagLogisticGrowth` models bacterial growth using a two population system (Yates and Smotzer, 2007).  $N$  represents bacteria that divide and  $N'$  represent bacteria that do not divide and eventually die. A transition between the two states is included to model bacteria that are initially non dividing but after a delay they start dividing. After the cross-cutting concern is included the model becomes



## 6 Post-translational coupling of a bistable switch and an oscillator

### 6.1 Code for the combined and coupled systems

The code for the combined system where proteins AraC and A are tagged is shown below.

Listing 12: Post-translational coupling of a bistable with and an oscillator. Coupling proteins AraC and A

---

```

1 declareNewMolecule('GFP')
2 declareNewMolecule('LacI')
3 declareNewMolecule('AraC_tag')
4 declareNewMolecule('LacIn4')
5 declareNewMolecule('AraCn2')
6 declareNewMolecule('zero')
7
8 declareNewPart("P1", HybridPromoter, [LacIn4, AraCn2], regulatorInfoMap={LacIn4:False,
    AraCn2:True})

```

```

9  declareNewPart("P2", HybridPromoter, [LacIn4, AraCn2], regulatorInfoMap={LacIn4:False,
    AraCn2:True})
10 declareNewPart("P3", HybridPromoter, [LacIn4, AraCn2], regulatorInfoMap={LacIn4:False,
    AraCn2:True})
11
12 declareNewMolecule('A_tag')
13 declareNewMolecule('B')
14 declareNewMolecule('An2')
15 declareNewMolecule('Bn2')
16 declareNewMolecule('In')
17
18 declareNewPart('Pa', NegativePromoter, [An2])
19 declareNewPart('Pb', NegativePromoter, [Bn2])
20
21 class S(Circuit):
22     def mainCircuit(self):
23
24         self.createMolecule(GFP)
25         self.createMolecule(LacI)
26         self.createMolecule(AraC_tag)
27         self.createMolecule(LacIn4)
28         self.createMolecule(AraCn2)
29
30         self.addPart(P1)
31         self.addPart(CodingRegion(AraC_tag))
32
33         self.addPart(P2)
34         self.addPart(CodingRegion(LacI))
35
36         self.addPart(P3)
37         self.addPart(CodingRegion(GFP))
38
39         self.reactionFrom(LacI, LacI, LacI, LacI) >> self.reactionTo( LacIn4 )
40         self.reactionFrom(LacIn4) >> self.reactionTo( zero )
41         self.reactionFrom(AraC_tag, AraC_tag) >> self.reactionTo( AraCn2 )
42         self.reactionFrom(AraCn2) >> self.reactionTo( zero )
43
44 class T(Circuit):
45     def mainCircuit(self):
46
47         self.createMolecule(A_tag)
48         self.createMolecule(An2)
49         self.createMolecule(B)
50         self.createMolecule(Bn2)
51         self.createMolecule(In)
52
53         self.addPart(Pa)
54         self.addPart(CodingRegion(B))
55         self.addPart(Pb)
56         self.addPart(CodingRegion(A_tag))
57
58         self.reactionFrom(A_tag, A_tag) >> self.reactionTo( An2 )
59         self.reactionFrom(B, B) >> self.reactionTo( Bn2 )
60         self.reactionFrom(An2) >> self.reactionTo( zero )
61         self.reactionFrom(Bn2) >> self.reactionTo( zero )

```

```

62
63         self.reactionFrom(zero) >> self.reactionTo( In )
64         self.reactionFrom(In,Bn2) >> self.reactionTo( In )
65
66 class toggleOsc(Circuit):
67     def mainCircuit(self):
68         self.addCircuit(S)
69         self.addCircuit(T)
70
71
72 compiledDesign = Weaver(toggleOsc, DesignRules, PromoterMapping, SheaAckersKineticsRNA,
73                         PostTranslationalCoupling, PrintReactionNetwork).output()
74 compiledDesign.printReactionNetwork()

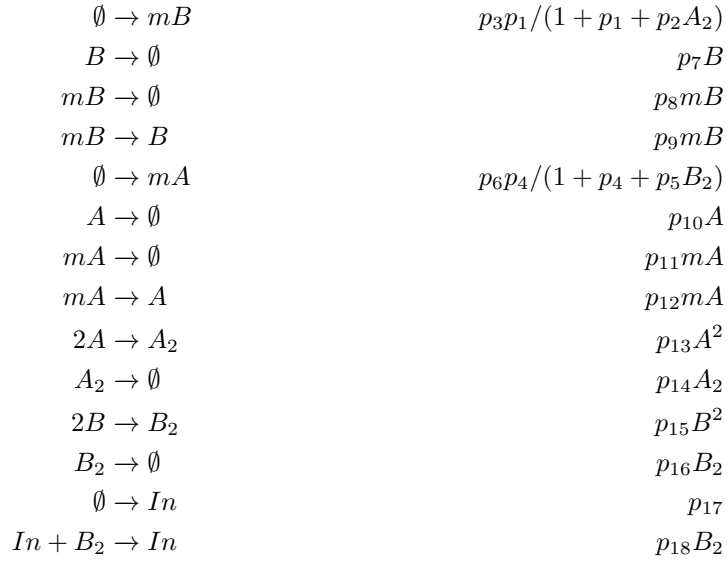
```

---

The models, given below, are generated using the `SheaAckersKineticsRNA` aspect. The aspect `PostTranslationalCoupling` identifies proteins with names ending in `_tag` and assigns them an enzymatic degradation term.

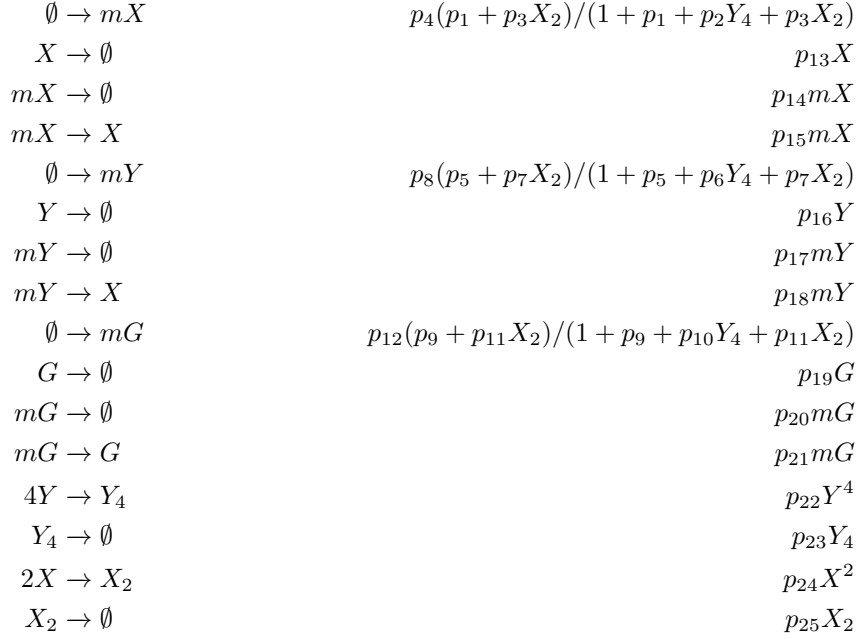
## 6.2 Toggle switch model

The numbers of the the two proteins, A and B, are represented by  $A$  and  $B$  respectively and their mRNA transcripts by  $mA$  and  $mB$ . We assume that both proteins form dimers. The number of inducer molecules is represented by  $In$ .



### 6.3 Stricker oscillator model

The numbers of the AraC, LacI and GFP proteins are represented by  $X$ ,  $Y$  and  $G$  respectively and their mRNA transcripts by  $mX$ ,  $mY$  and  $mG$ .



## References

- M. B. Elowitz and S. Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767): 335–338, Jan. 2000. doi: 10.1038/35002125.
- C. F. Lopez, J. L. Muhlich, J. A. Bachman, and P. K. Sorger. Programming biological models in Python using PySB. *Molecular Systems Biology*, 9:646, 2013. doi: 10.1038/msb.2013.1.
- D. Stewart and J. R. Wilson-Kanamori. Modular modelling in synthetic biology: Light-based communication in e. coli. *Electr. Notes Theor. Comput. Sci.*, 277:77–87, 2011.
- B. Wang, R. I. Kitney, N. Joly, and M. Buck. Engineering modular and orthogonal genetic logic gates for robust digital-like synthetic biology. *Nature Communications*, 2:508, 2011.
- G. T. Yates and T. Smotzer. On the lag phase and initial decline of microbial growth curves. *Journal of theoretical biology*, 244(3):511–517, Feb. 2007. doi: 10.1016/j.jtbi.2006.08.017.