Programming biological models in Python using PySB

Supplementary Information

Carlos F. Lopez[1,2,3], Jeremy L. Muhlich[1,2], John A. Bachman[1,2] and Peter K. Sorger[2,4]

[1] Authors contributed equally to this work

[2] Center for Cell Decision Processes, Harvard Medical School, Department of Systems Biology, 200 Longwood Ave, Boston MA 02115, USA

[3] Current address: Department of Cancer Biology, Center for Quantitative Sciences, Vanderbilt University School of Medicine, 2220 Pierce Avenue, Nashville TN 37232-6848, USA

[4] To whom correspondence should be addressed

Email: Carlos Lopez - c.lopez@vanderbilt.edu; Jeremy Muhlich - jeremy_muhlich@hms.harvard.edu; John Bachman - bachman@fas.harvard.edu; Peter Sorger - peter_sorger@hms.harvard.edu;

Contents

Supplementary note

In examining the differential equations for the models in Cui et al. [1], we found the following two errors.

Misplaced consumption term for activator

The reaction for the activation of Bax is described in Table 1 of [1] ("Chemical reaction network scheme") as

$$\text{InBax} + \text{Act} \rightarrow \text{AcBax} + \text{Act}$$

In the list of ODEs provided in Table 2, the term describing the velocity of this reaction is listed (correctly) as

$$J_1 = k_1 \cdot [\text{InBax}] \cdot [\text{Act}]$$

The term J_1 appears in the following equations shown in Table 2 (some terms omitted for clarity):

$$\begin{aligned} d[\text{InBax}]/dt &= J_{InBax} - J_1 - J_9 \\ d[\text{AcBax}]/dt &= J_{AcBax} + J_1 - J_2 - \ldots \\ d[\text{Act}]/dt &= J_{Act} - J_1 - J_3 + \ldots \end{aligned}$$

The error is that J_1 appears as a negative term in the equation for [Act], implying that activator is actually consumed in the reaction $\text{InBax} + \text{Act} \rightarrow \text{AcBax} + \text{Act}$. However, because this is a one-step "catalytic" reaction, with no intermediate complex formed, activator should not be consumed.

An additional indication that this an error is that in the equations for the "Direct" model from Chen et al. [2], on which the Cui et al. [1] models are based, the equation for Activator does not have this negative term. The appearance of this error in the derived model, but not the original model, highlights the tendency of "copy-and-paste" model reuse to introduce inadvertent errors.

Missing term for Bax inactivation

The reaction for Bax inactivation is listed in Table 1 of Cui et al. [1] ("Chemical reaction network scheme") as

$$\text{AcBax} \rightarrow \text{InBax}$$

In the list of ODEs provided in Table 2, the term describing the velocity of this reaction is J_5:

$$J_5 = k_8 \cdot [\text{AcBax}]$$

J_5 appears correctly as a negative term in the equation for activated Bax:

$$d[\text{AcBax}]/dt = J_{AcBax} + J_1 - J_2 - J_4 - J_5 + J_8 - J_9 - 2 \cdot J_{10}$$

However, the equation for inactive Bax omits J_5 completely, where it should be incorporated as a positive term:

$$d[\text{InBax}]/dt = J_{InBax} - J_1 - J_9$$

This means that active Bax is consumed with the rate defined by J_5, but the corresponding quantity of inactive Bax is not restored, leading to a loss of Bax over time. As with the error in the equation for Activator described above, the equation for Bax is correct in the original model from Chen et al. [2], indicating that the error was likely introduced in the process of duplicating the original model, or in the transcription of ODEs for publication.

PySB syntax

PySB model definition statements support some idiomatic syntax elements which may appear unusual to a Python programmer, but which provide economy of expression and a close mapping to the syntax of established rule-based modeling languages BNGL and Kappa. PySB defines overloaded operators and a so-called "SelfExporter" system which help give PySB model definition statements the feel of a domain-specific language like BNGL or Kappa within the general-purpose Python programming langage. Modelers are not required to use the overloaded operators and SelfExporter functionality, but they are highly recommended for ease of readability.

Overloaded operators simplify rule expression syntax

Python's built-in operators generally work with just a small set of built-in types, and an exception will be raised if they are applied to objects of any other class. For example "+" applies to numeric types (implementing mathematical addition), and separately to lists and strings (implementing concatenation). However a class may implement several special methods to *overload* these operators and allow them to be applied to its instances. A class is free to define any desired semantics in its overloaded operator implementations, but operator precedence and arity is fixed by the Python language grammar and cannot be altered via operator overloading.

PySB defines several classes representing nodes in an abstract syntax tree (AST) representation of a rule in the BNGL/Kappa domain-specific languages (DSL). These classes define overloaded operators allowing a modeler to write a Python expression which is visually very similar to one of these rules and evaluates to the corresponding AST. Whereas BNGL and Kappa use a standalone program to parse and simulate a model defined in a DSL, The PySB AST classes with operator overloading allow model components to be declared with Python program statements which evaluate directly to object representations of the given components. In this way Python itself serves as the parser for PySB models, and model components become live Python objects. The following table lists the AST classes and the syntactic elements they represent:

Class	Description
MonomerPattern	A pattern which matches instances of a given monomer, with optional restrictions on the state of certain sites.
ComplexPattern	A bound set of MonomerPatterns, i.e. a pattern to match a complex.
ReactionPattern	A pattern for the entire product or reactant side of a rule.
RulePattern	A container for the reactant and product patterns of a rule expression.

The operators to overload were chosen to fulfill two requirements. First, they must have the proper relative precedence in the Python grammar to minimize the need for parentheses and keep rule expressions uncluttered. Second, they should have a visual appearance as close as possible to the operators used in BNG and Kappa to help maintain consistency within the rule-based modeling ecosystem. The following table lists the overloaded operators and their semantics:

Operator	**Description**
( )	Apply site conditions to a Monomer to create a MonomerPattern
%	Combine MonomerPatterns to create a ComplexPattern
+	Combine ComplexPatterns to create a ReactionPattern
<>	Combine two ReactionPatterns to create a reversible RulePattern
>>	Combine two ReactionPatterns to create an irreversible RulePattern

For those who are familiar with BioNetGen Language (BNGL), here are some actual expressions in both PySB and BNG syntax (Kappa is similar to BNGL):

PySB expression	**BNGL equivalent**
R(a=None)	R(a)
R(a=1) % R(b=1)	R(a!1).R(b!1)
R(a=None) + R(b=None)	R(a) + R(b)
R(a=None) + R(b=None) <> R(a=1) % R(b=1)	R(a) + R(b) <-> R(a!1).R(b!1)
R(a=None) + R(b=None) >> R(a=1) % R(b=1)	R(a) + R(b) -> R(a!1).R(b!1)

Below is a formal grammar for PySB rule expressions. Symbols corresponding to the Python AST node classes are shown in bold, using the actual class name. Symbols which are self-explanatory such as "site-name" and "string" are not expanded further.

⟨**MonomerPattern**⟩ → ⟨*monomer*⟩ **'('** ⟨*site-conditions*⟩ **')'**

⟨*site-conditions*⟩ → ⟨*site-name*⟩ **'='** ⟨*condition*⟩ **','** ⟨*site-conditions*⟩
| ⟨*site-name*⟩ **'='** ⟨*condition*⟩
| ϕ

⟨*condition*⟩ → ⟨*string*⟩
| ⟨*bond-number*⟩
| **'('** ⟨*bond-number-list*⟩ **')'**
| **'('** ⟨*string*⟩ **','** ⟨*bond-number*⟩ **')'**

⟨*bond-number-list*⟩ → ⟨*bond-number*⟩ **','** ⟨*bond-number*⟩
| ⟨*bond-number-list*⟩ **','** ⟨*bond-number*⟩

⟨**ComplexPattern**⟩ → ⟨**ComplexPattern**⟩ **'%'** ⟨**ComplexPattern**⟩
| ⟨**MonomerPattern**⟩

⟨**ReactionPattern**⟩ → ⟨**ReactionPattern**⟩ **'+'** ⟨**ReactionPattern**⟩
| ⟨**ComplexPattern**⟩

⟨**RulePattern**⟩ → ⟨**ReactionPattern**⟩ ⟨*rule-op*⟩ ⟨**ReactionPattern**⟩

⟨*rule-op*⟩ → **'<>'** | **'>>'**

Excessive or haphazard use of operator overloading can certainly lead to confusing code, but we felt the construction of rule ASTs was a reasonable application with a limited scope. For comparison, here are several PySB rule expressions written using both overloaded operators and explicit AST assembly. The explicit forms of the first four subexpressions look simple enough in isolation, but the economy of the overloaded operators becomes readily apparent upon considering the final full RuleExpression.

Operators	**Explicit AST assembly**
mp = R(a=1)	mp = MonomerPattern(R, {'a': 1})
cp = mp1 % mp2	cp = ComplexPattern([mp1, mp2])
rp = cp1 + cp2	rp = ReactionPattern([cp1, cp2])
re = rp1 <> rp2	re = RuleExpression(rp1, rp2, True)

R(a=None) + R(a=None) <> R(a=1) % R(a=1)	RuleExpression(
	  ReactionPattern([
	  ComplexPattern([
	  MonomerPattern(R, {'a': None})
	  ]),
	  ComplexPattern([
	  MonomerPattern(R, {'a': None})
	  ])
	  ]),
	  ReactionPattern([
	  ComplexPattern([
	  MonomerPattern(R, {'a': 1}),
	  MonomerPattern(R, {'a': 1})
	  ])
	  ]),
	  True
	)

`SelfExporter` functionality streamlines model construction

PySB also includes functionality to streamline the process of creating components and adding them to models, using a class called `SelfExporter`. Like all object constructors in Python, each of the component constructors (`Monomer, Rule, Parameter` and `Compartment`) return an instance of the requested component. In a typical programming paradigm, it would be necessary to explicitly retain a reference to the created object in a variable for later use. For example, creating a monomer "R" and parameter "kf" for use in a rule declaration would require the following statements:

```
R = Monomer('R', ['a'])
kf = Parameter('kf', 1)
dimerize = Rule('dimerize', R(a=None) + R(a=None) >> R(a=1) % R(a=1), kf)
```

Here the `Monomer` constructor is used to create an instance of a `Monomer` object named "R", stored in the local variable `R`. From a modeling perspective, one can immediately see a potentially confusing aspect of this approach: we now have to mentally keep track of two "names" for the same monomer, one the variable storing the reference to the object (`R`, which must be used to

build up the expression for the dimerization rule) and one the descriptive name assigned to the new object ("R"). Even though we have chosen to use the same name for both the object's descriptive name and its variable in order to minimize confusion, maintaining this consistency requires mental effort on the part of the modeler and clutters the code making it harder to read.

In addition to managing the issue of naming, we must also add the newly created `Monomer`, `Parameter` and `Rule` objects to a model. To do this, we must call the `model.add_component` method on each component object:

```
model = Model('model')
model.add_component(R)
model.add_component(kf)
model.add_component(dimerize)
```

This repetition adds further visual noise to the model code, and accidentally omitting the `add_component` call for one or several components can lead to errors far from the site of declaration (in the case of a Monomer or Parameter used in a distant Rule) or worse, subtle errors in model behavior (in the case of a Rule).

In a typical modeling scenario, creation of model components tends to follow the pattern described above, that is:

1. Create a component using the appropriate constructor and assign it to a variable in the current namespace.
2. Add the created component to the current model.

The repetition of this pattern for every component in a model tends to be verbose and obscure the model structure; it also creates opportunities for error as described above.

PySB includes a helper class called `SelfExporter` that streamlines model definition by automatically performing the above steps. Using the functionality provided behind the scenes by `SelfExporter`, we can now simply write:

```
Model('model')
Monomer('R', ['a'])
Parameter('kf', 1)
Rule('dimerize', R(a=None) + R(a=None) >> R(a=1) % R(a=1), kf)
```

In the above example, when the constructor `Model('model')` is called, the `SelfExporter` "exports" a reference to the model by *creating a global variable called `model` in the current namespace and assigning a reference to the created `Model` object to it.* (This is possible because by using the Python module `inspect`, global variables in any accessible namespace can be accessed and manipulated programmatically: they are stored as a dictionary linking the name of the variable ("model") to its value (a reference to the new `Model` object). The `SelfExporter` can add global variables by modifying the entries in this dictionary.)

When the second statement, `Monomer('R', ['a'])`, is executed, the `SelfExporter` performs a similar action: it creates a new variable, `R`, and assigns to it the reference to the new object (a `Monomer` object given the name "R"). However, in this case it also takes a second action: it adds the new `Monomer` object `R` to the set of `Monomer` objects associated with the currently defined model, `model`. The process for the `Parameter` object is exactly the same: a global variable `kf` for the new object is created and added to `model`.

Finally, because they have been "exported" as variables by the behind-the-scenes action of the `SelfExporter`, the `Monomer` variable `R` and the `Parameter` variable `kf` are now globally accessible, and we can use both in the `Rule(...)` definition that follows. The `Rule` object itself is similarly exported and added to the model.

To summarize, the execution of the above code results in:

- The addition of four variables to the global namespace: `model`, `R`, `kf`, and `dimerize`
- The addition of the model components `R`, `kf`, and `dimerize` to the model `model`.

It should be noted that though it is the default behavior, *the use of `SelfExporter` functionality is entirely optional.* In certain sophisticated modeling scenarios involving the dynamic creation of multiple alternative models, the explicit approach to component creation and assignment may be preferred. However, we have found that the `SelfExporter` substantially simplifies the most common modeling use cases. A summary of the syntax for the simple example described above, with and without the action of `SelfExporter`, is shown below:

With SelfExporter	**Without SelfExporter**
Model('model')	model = Model('model')
Monomer('R', ['a'])	R = Monomer('R', ['a']) model.add_component(R)
Parameter('kf', 1)	kf = Parameter('kf', 1) model.add_component(kf)
Rule('dimerize', R(a=None) + R(a=None) >> R(a=1) % R(a=1), kf)	dimerize = Rule('dimerize', R(a=None) + R(a=None) >> R(a=1) % R(a=1), kf) model.add_component(dimerize)

The following comparison with the BNGL syntax for the same simple model shows how the combined use of overloaded operators and the `SelfExporter` give models written in PySB the feel of a domain-specific language embedded in Python:

PySB statement	**BNGL equivalent**
Model('model')	*(not needed)*
Monomer('R', ['a'])	begin molecule types R(a) end molecule types
Parameter('kf', 1)	begin parameters kf 1 end parameters

Rule('dimerize',
  R(a=None) + R(a=None) >> R(a=1) % R(a=1),
  kf)
	begin reaction rules
	  dimerize: R(a) + R(a) -> R(a!1).R(a!1) kf
	end reaction rules

Figure S1

A) `catalyze` macro basic implementation

Basic Implementation

```
def catalyze(enz, e_site, sub, s_site, prod, klist):
    kf, kr, kc = klist   # Get the parameters from the list

    # Create the rules
    rb = Rule('bind_%s_%s' % (enz().monomer.name, sub().monomer.name),
           enz({e_site:None}) + sub({s_site:None}) <>
           enz({e_site:1}) % sub({s_site:1}),
           kf, kr)
    rc = Rule('catalyze_%s%s_to_%s' %
           (enz().monomer.name, sub().monomer.name, prod().monomer.name),
           enz({e_site:1}) % sub({s_site:1}) >>
           enz({e_site:None}) + prod({s_site:None}),
           kc)
    return [rb, rc]
```

B) `catalyze_one_step` macro

Example Macro Call

```
catalyze_one_step(C8(bf=None), Bid(state='U', bf=None),
                  Bid(state='T', bf=None), kf)
```

Basic Implementation

```
def catalyze_one_step(enz, sub, prod, kf):
    # Create the rule
    r = Rule('catalyze_one_step_%s_%s_to_%s' %
          (enz.monomer.name, sub.monomer.name, prod.monomer.name),
          enz() + sub() >> enz() + prod(),
          kf)
    return r
```

BNGL Rules

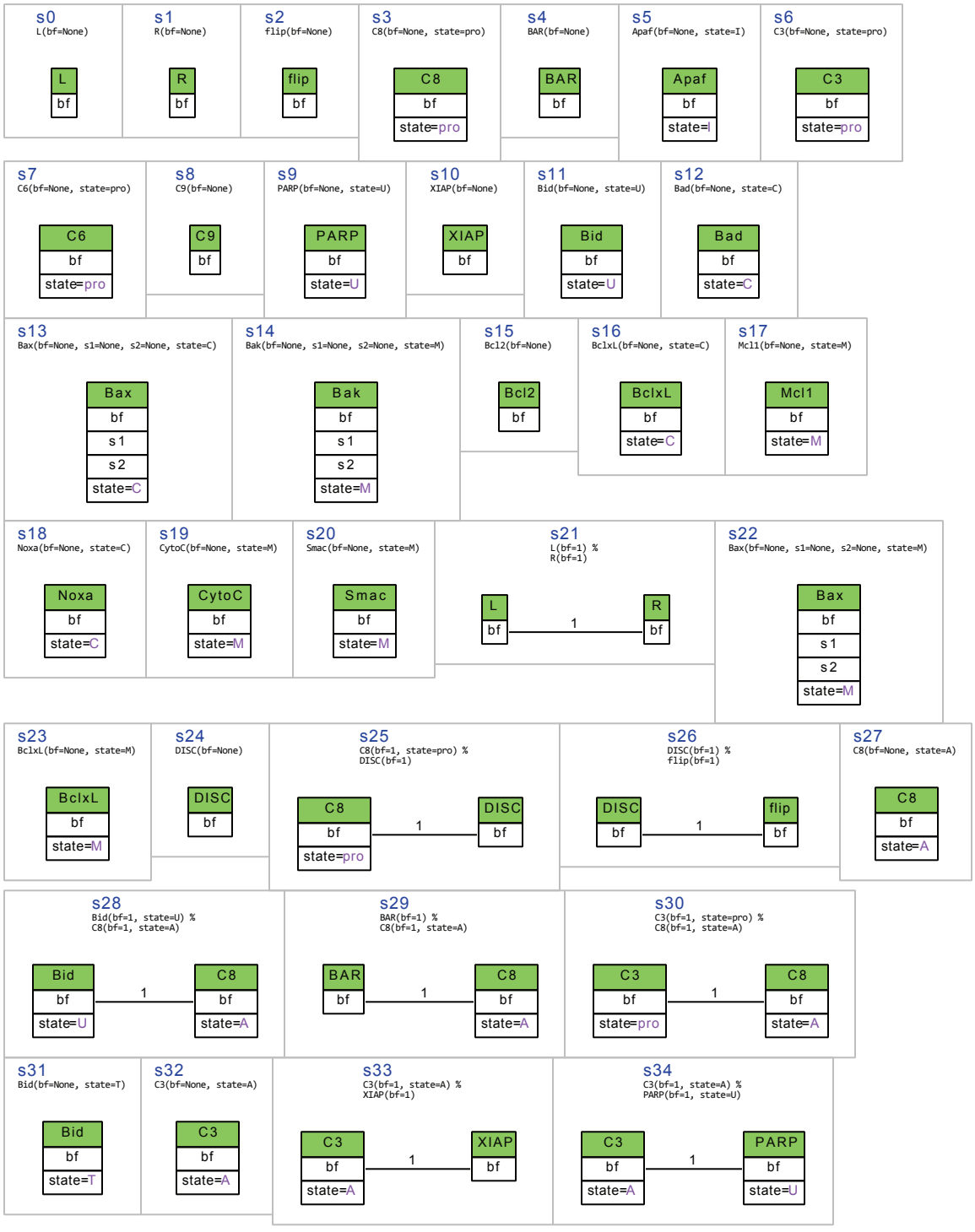
```
C8(bf) + Bid(bf,state~U) -> C8(bf) + Bid(bf,state~T)  kf
```

ODEs

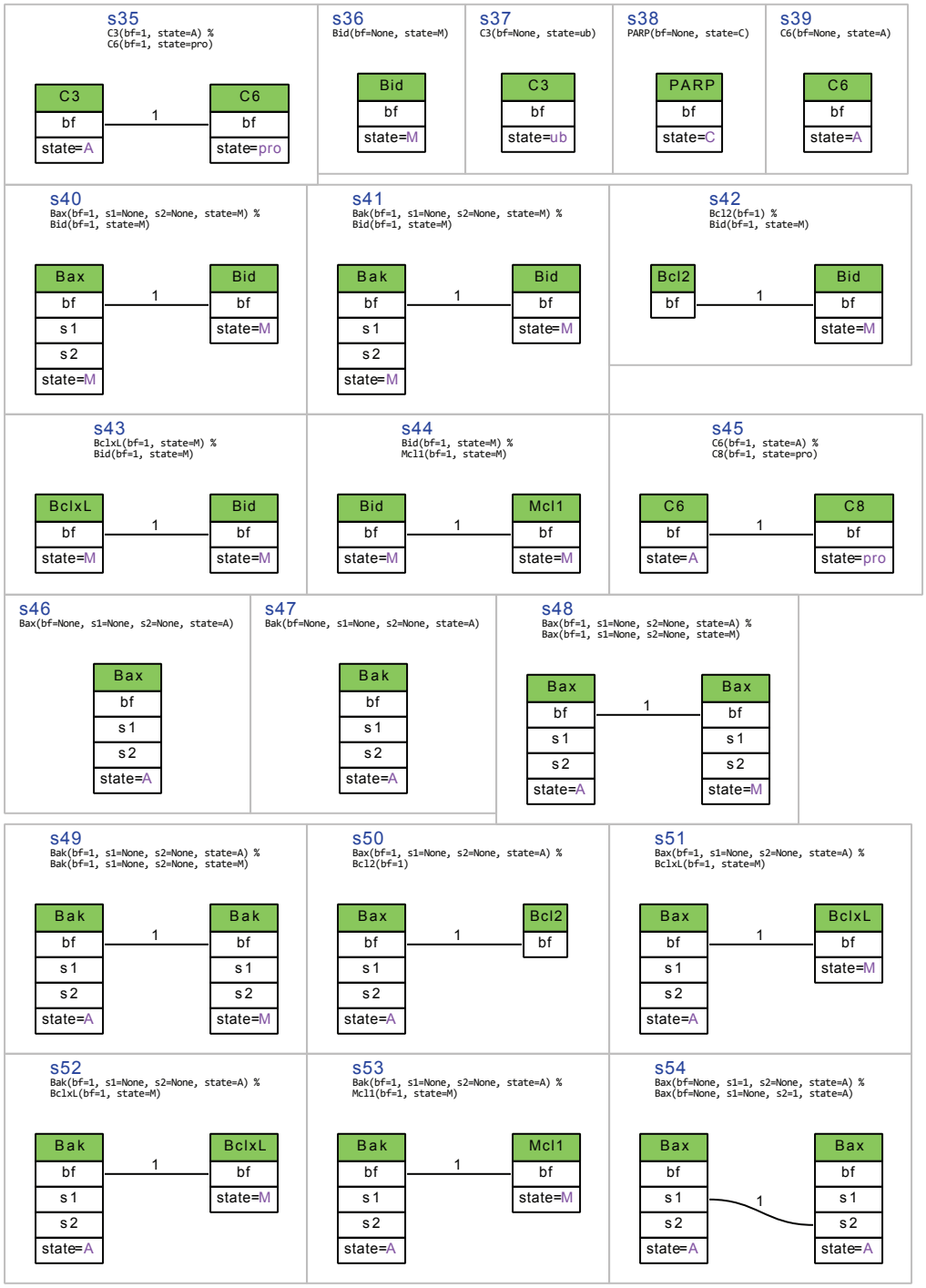
```
C8:   ds0/dt =  0
Bid:  ds1/dt = -kf*s0*s1
tBid: ds2/dt =  kf*s0*s1
```

(A) Simplified implementation of the `catalyze` macro. The `Rule` objects for the binding and catalytic steps are created according to defined templates, with the species identities (enzyme, substrate, and product), binding site names, and parameters filled in from the arguments to the macro. **(B)** `catalyze_one_step`. This macro models a "one-step" approximation of catalysis according to the reaction scheme $E+S \rightarrow E+P$. The macro creates a single catalysis rule according to the prescribed template, which can then be used to generate the BNGL rule and set of ODEs shown below. The full implementation of the `catalyze` and `catalyze_one_step` macros, with documentation and handling of various special cases, can be found in the `macros.py` file in the PySB source code online (http://pysb.org).

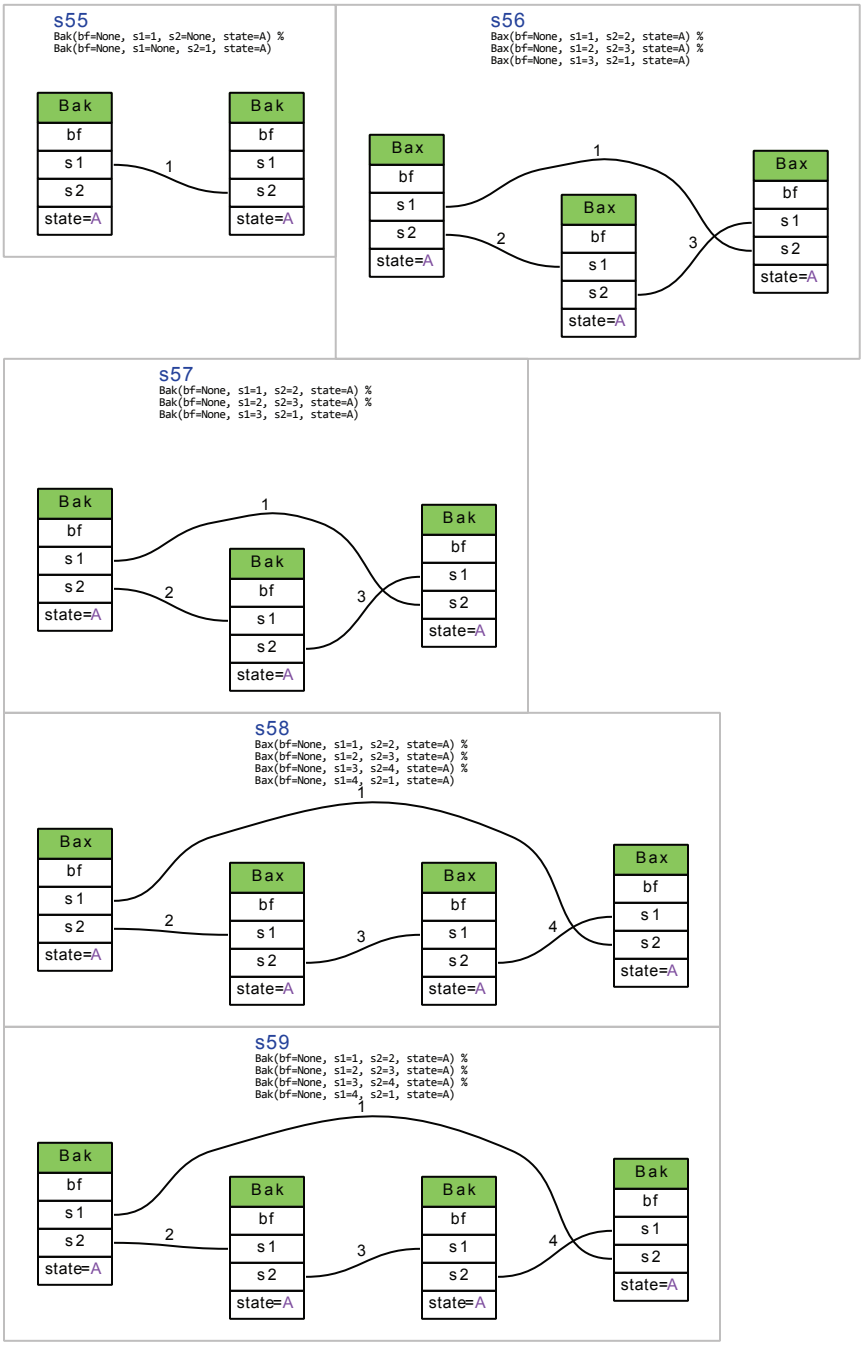
Figure S2



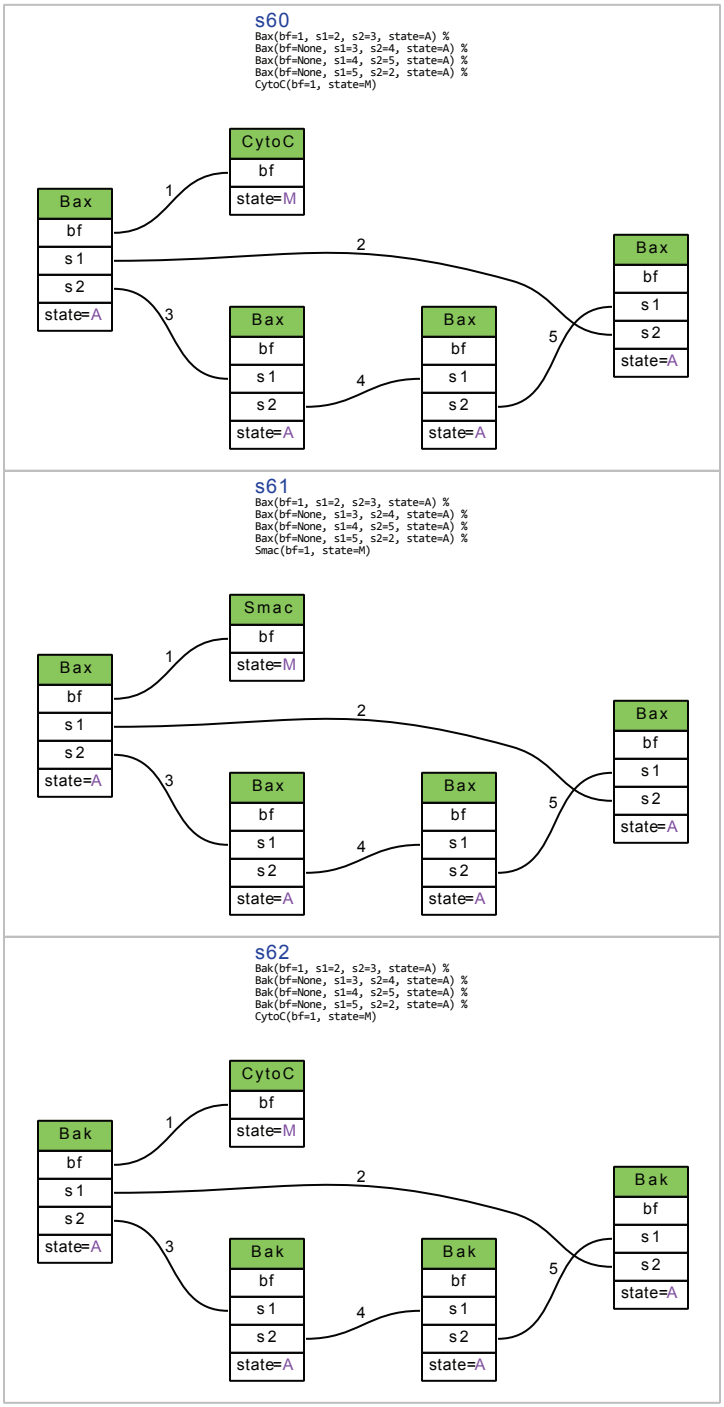
continued...


s35
C3(bf=1, state=A) %
C6(bf=1, state=pro)
C3
bf
state=A
1
C6
bf
state=pro
s36
Bid(bf=None, state=M)
Bid
bf
state=M
s37
C3(bf=None, state=ub)
C3
bf
state=ub
s38
PARP(bf=None, state=C)
PARP
bf
state=C
s39
C6(bf=None, state=A)
C6
bf
state=A
s40
Bax(bf=1, s1=None, s2=None, state=M) %
Bid(bf=1, state=M)
Bax
bf
s1
s2
state=M
1
Bid
bf
state=M
s41
Bak(bf=1, s1=None, s2=None, state=M) %
Bid(bf=1, state=M)
Bak
bf
s1
s2
state=M
1
Bid
bf
state=M
s42
Bcl2(bf=1) %
Bid(bf=1, state=M)
Bcl2
bf
1
Bid
bf
state=M
s43
BclxL(bf=1, state=M) %
Bid(bf=1, state=M)
BclxL
bf
state=M
1
Bid
bf
state=M
s44
Bid(bf=1, state=M) %
Mcl1(bf=1, state=M)
Bid
bf
state=M
1
Mcl1
bf
state=M
s45
C6(bf=1, state=A) %
C8(bf=1, state=pro)
C6
bf
state=A
1
C8
bf
state=pro
s46
Bax(bf=None, s1=None, s2=None, state=A)
Bax
bf
s1
s2
state=A
s47
Bak(bf=None, s1=None, s2=None, state=A)
Bak
bf
s1
s2
state=A
s48
Bax(bf=1, s1=None, s2=None, state=A) %
Bax(bf=1, s1=None, s2=None, state=M)
Bax
bf
s1
s2
state=A
1
Bax
bf
s1
s2
state=M
s49
Bak(bf=1, s1=None, s2=None, state=A) %
Bak(bf=1, s1=None, s2=None, state=M)
Bak
bf
s1
s2
state=A
1
Bak
bf
s1
s2
state=M
s50
Bax(bf=1, s1=None, s2=None, state=A) %
Bcl2(bf=1)
Bax
bf
s1
s2
state=A
1
Bcl2
bf
s51
Bax(bf=1, s1=None, s2=None, state=A) %
BclxL(bf=1, state=M)
Bax
bf
s1
s2
state=A
1
BclxL
bf
state=M
s52
Bak(bf=1, s1=None, s2=None, state=A) %
Bclxl(bf=1, state=M)
Bak
bf
s1
s2
state=A
1
BclxL
bf
state=M
s53
Bak(bf=1, s1=None, s2=None, state=A) %
Mcl1(bf=1, state=M)
Bak
bf
s1
s2
state=A
1
Mcl1
bf
state=M
s54
Bax(bf=None, s1=1, s2=None, state=A) %
Bax(bf=None, s1=None, s2=1, state=A)
Bax
bf
s1
s2
state=A
1
Bax
bf
s1
s2
state=A

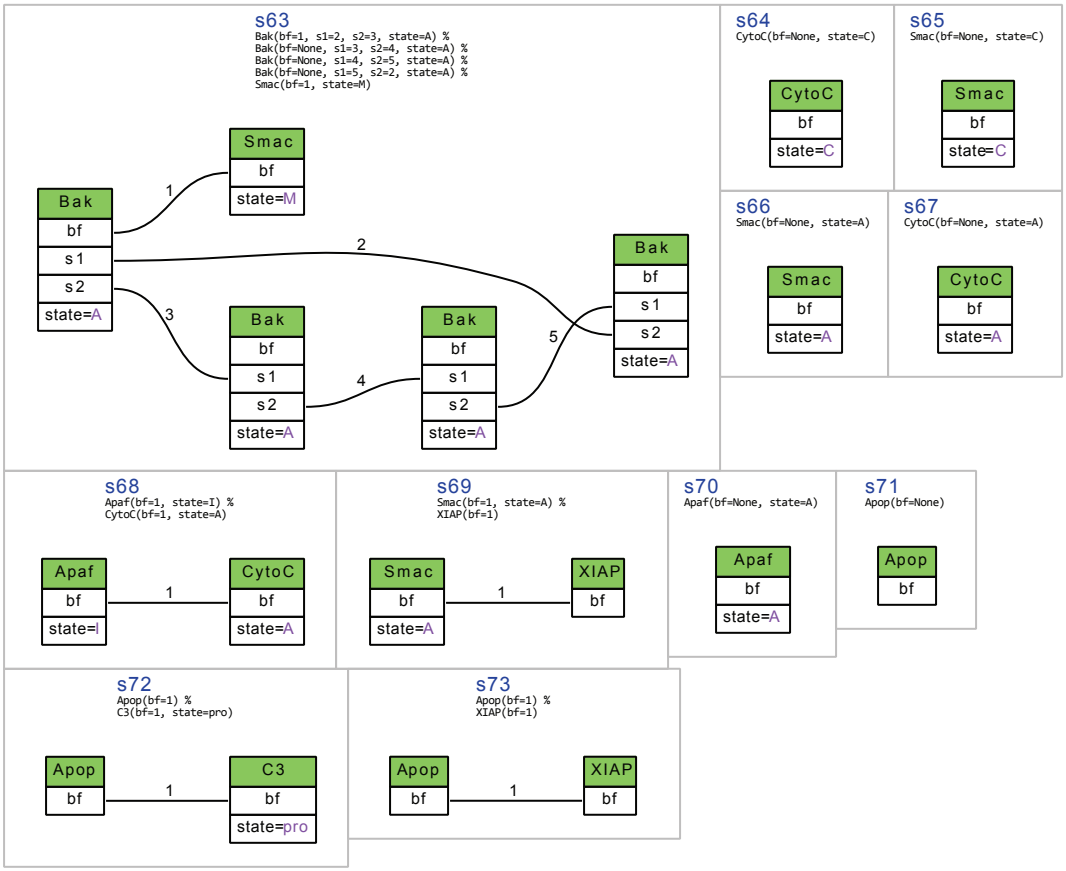
continued...


s55
Bak(bf=None, s1=1, s2=None, state=A) %
Bak(bf=None, s1=None, s2=1, state=A)
Bak
bf
s1
s2
state=A
1
s56
Bax(bf=None, s1=1, s2=2, state=A) %
Bax(bf=None, s1=2, s2=3, state=A) %
Bax(bf=None, s1=3, s2=1, state=A)
Bax
bf
s1
s2
state=A
1
2
3
s57
Bak(bf=None, s1=1, s2=2, state=A) %
Bak(bf=None, s1=2, s2=3, state=A) %
Bak(bf=None, s1=3, s2=1, state=A)
s58
Bax(bf=None, s1=1, s2=2, state=A) %
Bax(bf=None, s1=2, s2=3, state=A) %
Bax(bf=None, s1=3, s2=4, state=A) %
Bax(bf=None, s1=4, s2=1, state=A)
4
s59
Bak(bf=None, s1=1, s2=2, state=A) %
Bak(bf=None, s1=2, s2=3, state=A) %
Bak(bf=None, s1=3, s2=4, state=A) %
Bak(bf=None, s1=4, s2=1, state=A)

continued...

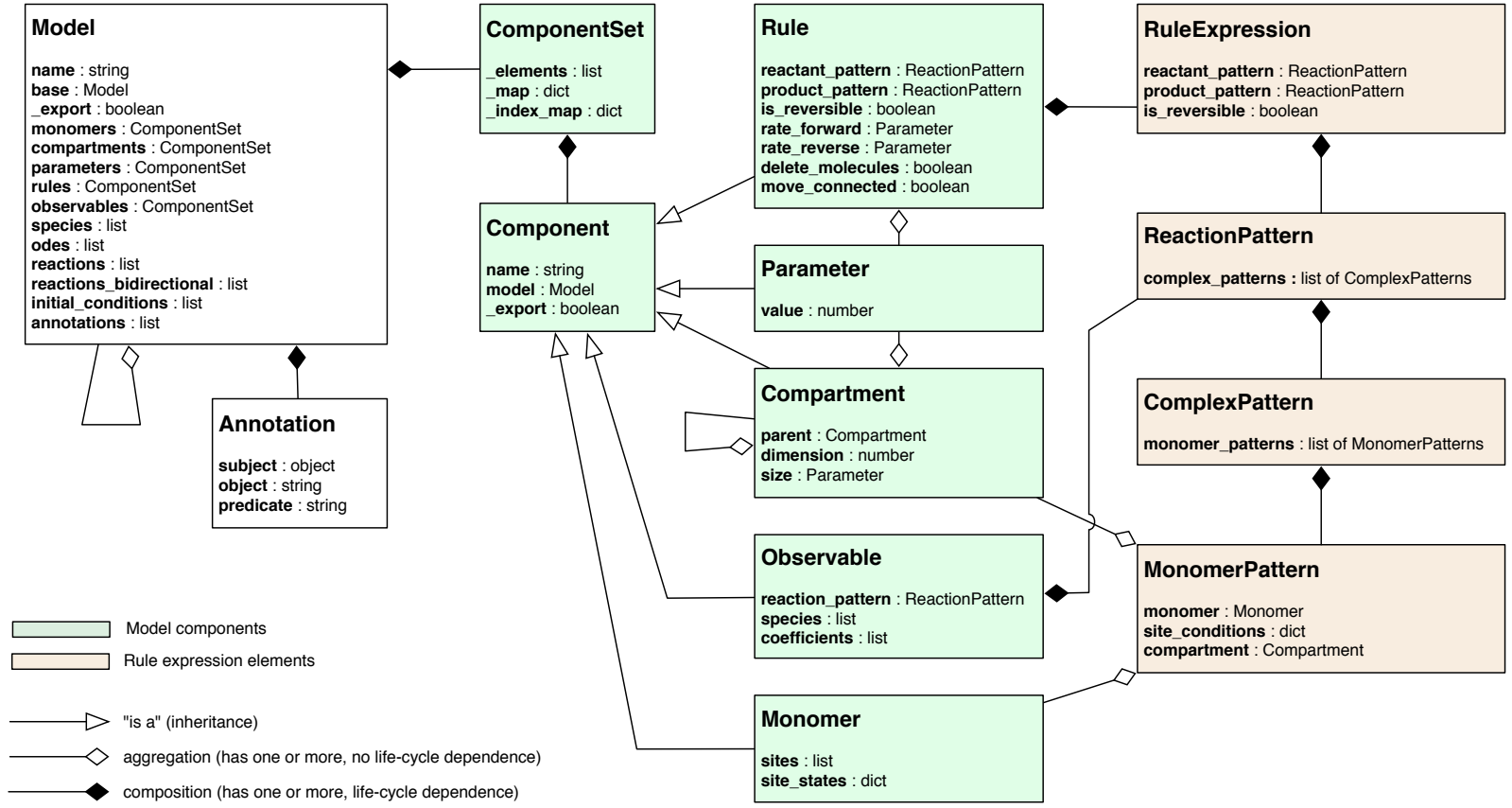

s60
Bax(bf=1, s1=2, s2=3, state=A) %
Bax(bf=None, s1=3, s2=4, state=A) %
Bax(bf=None, s1=4, s2=5, state=A) %
Bax(bf=None, s1=5, s2=2, state=A) %
CytoC(bf=1, state=M)
CytoC
bf
state=M
Bax
bf
s1
s2
state=A
1
2
3
4
5
s61
Bax(bf=1, s1=2, s2=3, state=A) %
Bax(bf=None, s1=3, s2=4, state=A) %
Bax(bf=None, s1=4, s2=5, state=A) %
Bax(bf=None, s1=5, s2=2, state=A) %
Smac(bf=1, state=M)
Smac
bf
state=M
Bax
bf
s1
s2
state=A
1
2
3
4
5
s62
Bak(bf=1, s1=2, s2=3, state=A) %
Bak(bf=None, s1=3, s2=4, state=A) %
Bak(bf=None, s1=4, s2=5, state=A) %
Bak(bf=None, s1=5, s2=2, state=A) %
CytoC(bf=1, state=M)
CytoC
bf
state=M
Bak
bf
s1
s2
state=A
1
2
3
4
5

continued...



Output from the PySB `render_species` tool run against EARM 2.0-M1a (Lopez Embedded). Each large box represents one species, with its number and PySB representation at the top, followed by a depiction of the monomer graph. In the monomer graph, each segmented box represents a monomer, with its name in the first green segment and its sites in the following segments. Edges between monomer sites represent bonds, and their numeric labels correspond directly to the bond numbering in the PySB representation above.

Figure S3



UML class diagram for core PySB classes. Note that for clarity, only fields are shown (not methods). `Model` objects are built up using collections of `Rule`, `Parameter`, `Compartment`, `Observable`, and `Monomer` objects, which acquire generic functionality (such as a `name` field) from the superclass `Component`. `Component`s are aggregated in the collection class `ComponentSet`. Rules are expressed as instances of the class `RuleExpression`, which are built up using instances of the classes `ReactionPattern`, `ComplexPattern`, and `MonomerPattern`. The resulting `RuleExpression` is stored in a `Rule` object, along with the associated rate parameters (as `Parameter` objects). Additional information about model elements (including `Component`s or macros) can be stored in an `Annotation` object. The list of `Annotation`s is stored in the `Model`.

References

1. J. Cui, C. Chen, H. Lu, T. Sun, P. Shen, *PLoS ONE* **3**, e1469 (2008).
2. C. Chen, J. Cui, W. Zhang, P. Shen, *FEBS letters* **581**, 5143 (2007).