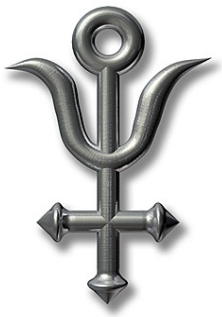
Antimony



A human-readable,
human-writable,
model definition language

Tutorial

Introduction
Since the advent of SBML (the Systems Biology Markup Language) computer models of biological systems have been able to be transferred easily between different labs and different computer programs without loss of specificity. But SBML was not designed to be readable or writable by humans, only by computer programs, so other programs have sprung up to allow users to more easily create the models they need.

Many of these programs are GUI-based, and allow drag-and-drop editing of species and reactions, such as JDesigner and TinkerCell. A few, like Jarnac, take a text-based approach, and allow the creation of models in a text editor. This has the advantage of being faster, more readily cross-platform, and readable by others without translation. Antimony (named this because the chemical symbol of the element is 'Sb') was designed as a successor to Jarnac's model definition language, with some new features that mesh with newer elements of SBML, some new features we feel will be generally applicable, and some new features that we hope will facilitate the creation of genetic networks in particular. A programming library 'libAntimony' was developed in tandem with the language to allow computer translation of Antimony-formatted models into SBML and other formats used by other computer programs.

The basic features of Antimony include the ability to:
- Simply define species and reactions,
- Package and re-use models as modules with defined or implied interfaces,
- Define events and compartments, and
- Create 'DNA strands' whose elements can pass reaction rates to downstream elements, and inherit and modify reaction rates from upstream elements.

Language Specifications

Species and Reactions

The simplest Antimony file may simply have a list of reactions containing species, along with some initializations. Reactions are written as two lists of species, separated by a '->', and followed by a semicolon:

```
S1 + E -> ES;
```

Optionally, you may provide a reaction rate for the reaction by including a mathematical expression after the semicolon, followed by another semicolon:

```
S1 + E -> ES; k1*k2*S1*E - k2*ES;
```

You may also give the reaction a name by prepending the name followed by a colon:

```
J0: S1 + E -> ES; k1*k2*S1*E - k2*ES;
```

The same effect can be achieved by setting the reaction rate separately, by assigning the reaction rate to the reaction name with an '=':

```
J0: S1 + E -> ES;
J0 = k1*k2*S1*E - k2*ES;
```

You may even define them in the opposite order—they are all ways of saying the same thing.

At this point, Antimony will make several assumptions about your model. It will assume (and require) that all symbols that appear in the reaction itself are species. Any symbol that appears elsewhere that is not used or defined as a species is 'undefined'; 'undefined' symbols may later be declared or used as species or as 'formulas', Antimony's term for constants and packaged equations like SBML's assignment rules. In the above example, k1 and k2 are (thus far) undefined symbols, which may be assigned straightforwardly:

```
J0: S1 + E -> ES; k1*k2*S1*E - k2*ES;
k1 = 3;
k2 = 1.4;
```

More complicated expressions are also allowed, as are the creation of symbols which exist only to simplify or clarify other expressions:

```
pH = 7;
k3 = -log10(pH);
```

The initial concentrations of species are defined in exactly the same way as formulas, and may be just as complex (or simple):

```
S1 = 2;
E = 3;
ES = S1 + E;
```

Order for any of the above (and in general in Antimony) does not matter at all: you may use a symbol before defining it, or define it before using it. As long as you do not use the same symbol in an incompatible context (such as using the same name as a reaction and a species), your resulting model will still be valid. Antimony files written by libAntimony will adhere to a standard format of defining symbols, but again, this is not required.

Constant and variable symbols

Some models have 'border species' in their reactions, or species whose concentrations remain constant. To declare something as constant, use the 'const' keyword as follows:

```
const S1;
```

While you're declaring it, you may want to be more specific by using the 'species' keyword:

```
const species S1;
```

If you have several species that are all constant, you may declare this all in one line:

```
const species S1, S2, S3;
```

While species are variable by default, you may also declare them so explicitly with the 'var' keyword:

```
var species S4, S5, S6;
```

Likewise, formulas are constant by default, unless their mathematical expression contains a variable symbol:

```
k1 = 5;
k2 = 2*S1;
```

k1 is constant by default, and k2 is variable by default if S1 is variable, and is constant by default if S1 is constant. Both defaults can be declared variable or constant explicitly:

```
const k1;
var k2;
```

You can also be more specific and declare that both are formulas:

```
const formula k1;
var formula k2;
```

If you declare that k2 is const, but use a variable symbol in its mathematical expression, Antimony will assume that your formula is the value of the symbol at the start of the simulation. In SBML terms, it will use the 'Initial Assignment' construct for const formulas, and the 'Assignment Rule' construct for variable formulas.

You can also mix-and-match your declarations however best suits what you want to convey:

```
species S1, S2, S3, S4;
formula k1, k2, k3, k4;

const S1, S4, k1, k3;
var S2, S3, k2, k4;
```

Antimony is a pure model definition language, meaning that all statements in the language serve to build a static model of a dynamic biological system. Unlike Jarnac, sequential programming techniques such as re-using a variable for a new purpose will not work:

```
pH = 7;
k1 = -log10(pH);
pH = 8.2;
k2 = -log10(pH);
```

In a sequential programming language, the above would result in different values being stored in k1 and k2. (This is how Jarnac works, for those familiar with that language/simulation environment.) In a pure model definition language like Antimony, 'pH', 'k1', and 'k2' are static symbols that are being defined by Antimony statements, and not processed in any way. A simulator that requests the mathematical expression for k1 will receive the string "-log10(pH)"; the same string it will receive for k2. A request for the mathematical expression for pH will receive the string "8.2", since that's the last definition found in the file. As such, k1 and k2 will end up being identical.

As a side note, we considered having libAntimony store a warning when presented with an input file such as the example above with a later definition overwriting an earlier definition. However, there was no way with our current interface to let the user know that a warning had been saved, and it seemed like there could be a number of cases where the user might legitimately want to override an earlier definition (such as when using submodules, as we'll get to in a bit). So for now, the above is valid Antimony input that just so happens to produce exactly the same output as:

```
pH = 8.2;
k1 = -log10(pH);
k2 = -log10(pH);
```

Modules

Antimony input files may define several different models, and may use previously-defined models as parts of newly-defined models. Each different model is known as a 'module', and is minimally defined by putting the keyword 'model' (or 'module', if you like) and the name you want to give the module at the beginning of the model definitions you wish to encapsulate, and putting the keyword 'end' at the end:

```
model example
  S + E -> ES;
end
```

After this module is defined, it can be used as a part of another model (this is the one time that order matters in Antimony). To import a module into another module, simply use the name of the module, followed by parentheses and a semicolon:

```
model example
  S + E -> ES;
end
```

```
model example2
  example();
end
```

This is usually not very helpful in and of itself—you'll likely want to give the submodule a name so you can refer to the things inside it. To do this, pre-pend a name followed by a colon:

```
model example2
  A: example();
end
```

Now, you can modify or define elements in the submodule by referring to symbols in the submodule by name, pre-pended with the name you've given the module, followed by a '.':

```
model example2
  A: example();
  A.S = 3;
end
```

This results in a model with a single reaction (`A.S + A.E -> A.ES`) and a single initial condition (`A.S = 3`).

You may also import multiple copies of modules, and modules that themselves contain submodules:

```
model example3
  A: example();
  B: example();
  C: example2();
end
```

This would result in a model with three reactions and a single initial condition.

```
  A.S + A.E -> A.ES
  B.S + B.E -> B.ES
  C.A.S + C.A.E -> C.A.ES
  C.A.S = 3;
```

You can also use the species defined in submodules in new reactions:

```
model example4
  A: example();
  A.S -> ; kdeg*A.S;
end
```

When combining multiple submodules, you can also 'attach' them to each other by declaring that a species in one submodule is the same species as is found in a different submodule by using the 'is' keyword (“A.S is B.S”). For example, let's say that we have a species which is known to bind reversibly to two different species. You could set this up as the following:

```
model side_reaction
  J0: S + E -> SE; k1*k2*S*E - k2*ES;
  S = 5;
  E = 3;
  SE = E+S;
  k1 = 1.2;
  k2 = 0.4;
end

model full_reaction
  A: side_reaction();
  B: side_reaction()
  A.S is B.S;
end
```

If you wanted, you could give the identical species a new name to more easily use it in the 'full_reaction' module:

```
model full_reaction
  var species S;
  A: side_reaction();
  B: side_reaction()
  A.S is S;
  B.S is S;
end
```

In this system, 'S' is involved in two reversible reactions with exactly the same reaction kinetics and initial concentrations. Let's now say the reaction rate of the second side-reaction takes the same form, but that the kinetics are twice as fast, and the starting conditions are different:

```
model full_reaction
  var species S;
  A: side_reaction();
  A.S is S;
  B: side_reaction()
  B.S is S;
  B.k1 = 2.4;
  B.k2 = 0.8;
  B.E = 10;
end
```

Note that since we defined the initial concentration of 'SE' as 'S + E', B.SE will now have a different initial concentration, since B.E has been changed.

Finally, we add a third side reaction, one in which S binds irreversibly, and where the complex it forms degrades. We'll need a new reaction rate, and a whole new reaction as well:

```
model full_reaction
  var species S;
  A: side_reaction();
  A.S is S;
  B: side_reaction()
  B.S is S;
  B.k1 = 2.4;
  B.k2 = 0.8;
  B.E = 10;
  C: side_reaction();
  C.S is S;
  C.J0 =  C.k1*C.k2*S*C.E
  J3: C.SE -> ; C.SE*k3;
  k3 = 0.02;
end
```

Note that defining the reaction rate of C.J0 used the symbol 'S; exactly the same result would be obtained if we had used 'C.S' or even 'A.S' or 'B.S'. Antimony knows that those symbols all refer to the same species, and will give them all the same name in subsequent output.

For convenience and style, modules may define an interface where some symbols in the module are more easily renamed. To do this, first enclose a list of the symbols to export in parentheses after the name of the model when defining it:

```
model side_reaction(S, k1)
  J0: S + E -> SE; k1*k2*S*E - k2*ES;
  S = 5;
  E = 3;
  SE = E+S;
  k1 = 1.2;
  k2 = 0.4;
end
```

Then when you use that module as a submodule, you can provide a list of new symbols in parentheses:

```
  A: side_reaction(spec2, k2);
```

is equivalent to writing:

```
  A: side_reaction();
  A.S is spec2;
  A.k1 is k2;
```

One thing to be aware of when using this method: Since wrapping definitions in a defined model is optional, all 'bare' declarations are defined to be in a default module with the name '__main'. If there are no unwrapped definitions, '__main' will still exist, but will be empty.

Other files

More than one file may be used to define a set of modules in Antimony through the use of the 'import' keyword. At any point in the file outside of a module definition, use the word 'import' followed by the name of the file in quotation marks, and Antimony will include the modules defined in that file as if they had been cut and pasted into your file at that point. SBML files may also be included in this way:

```
import "models1.txt"
import "oscli.xml"

model mod2()
  A: mod1();
  B: oscli();
end
```

In this example, the file 'models1.txt' is an Antimony file that defines the module 'mod1', and the file 'oscli.xml' is an SBML file that defines a model named 'oscli'. The Antimony module 'mod2' may then use modules from either or both of the other imported files.

Remember that imported files act like they were cut and pasted into the main file. As such, any bare declarations in the main file and in the imported files will all contribute to the default '__main' module (though no SBML file will contribute to this module).

Compartments

A compartment is a demarcated region of space that contains species and has a particular volume. In Antimony, you may ignore compartments altogether, and all species are assumed to be members of a default compartment with the imaginative name 'default_compartment' with a constant volume of 1. You may define other compartments by using the 'compartment' keyword:

```
  compartment comp1;
```

Compartments may also be variable or constant, and defined as such with 'var' and 'const':

```
  const compartment comp1;
  var compartment comp2;
```

The volume of a compartment may be set with an '=' in the same manner as species and reaction rates:

```
  comp1 = 5;
  comp2 = 3*S1;
```

To declare that something is in a compartment, the 'in' keyword is used, either during declaration:

```
  compartment comp1 in comp2;
  const species S1 in comp2;
  S2 in comp2;
```

or during assignment for reactions:

```
J0 in comp1: x -> y; k1*x;
y -> z; k2*y in comp2;
```

or submodules:

```
M0 in comp2: submod();
submod2(y) in comp3;
```

or other variables:

```
S1 in comp2 = 5;
```

Here are Antimony's rules for determining which compartment something is in:

-If the symbol has been declared to be in a compartment, it is in that compartment.
-If not, if the symbol is in a DNA strand (see the next section) which has been declared to be in a compartment, it is in that compartment. If the symbol is in multiple DNA strands with conflicting compartments, it is in the compartment of the last declared DNA strand that has a declared compartment in the model.
-If not, if the symbol is a member of a reaction with a declared compartment, it is in that compartment. If the symbol is a member of multiple reactions with conflicting compartments, it is in the compartment of the last declared reaction that has a declared compartment.
-If not, if the symbol is a member of a submodule with a declared compartment, it is in that compartment. If the symbol is a member of multiple submodules with conflicting compartments, it is in the compartment of the last declared submodule that has a declared compartment.
-If not, the symbol is in the compartment 'default_compartment', and is treated as having no declared compartment for the purposes of determining the compartments of other symbols.

Note that declaring that one compartment is 'in' a second compartment does not change the compartment of the symbols in the first compartment:

```
compartment c1, c2;
species s1 in c1, s2 in c1;
c1 in c2;
```

yields:

symbol	compartment
s1	c1
s2	c1
c1	c2
c2	default_compartment

Compartments may not be circular; “c1 in c2; c2 in c3; c3 in c1” is illegal.

Events

Events are discontinuities in model simulations that change the definitions of one or more symbols if certain conditions apply. The condition is expressed as a boolean formula, and the definition changes are expressed as assignments, using the keyword 'at' and the following syntax:

```
at<boolean expr.>: variable1=formula1: variable2=formula2 [etc];
```

such as:

```
at (x>5): y=3: x=2.3;
```

You may also give the event a name by prepending it with a colon:

```
E1: at(x>5): y=3: x=2.3;
```

(you may also claim an event is 'in' a compartment just like everything else ('E1 in comp1:'), but this declaration will never change the compartment of anything else.)

DNA Strands:

A new concept in Antimony that has not been modeled explicitly in previous model definition languages such as SBML is the idea of having DNA strands where downstream elements can inherit reaction rates from upstream elements. DNA strands are declared by connecting symbols with '--':

```
--P1--G1--stop--P2--G2--
```

You can also give the strand a name:

```
dna1: --P1--G1--
```

By default, the reaction rate or formula associated with an element of a DNA strand is equal to the reaction rate or formula of the element upstream of it in the strand. Thus, if P1 is a promoter and G1 is a gene, in the model:

```
dna1: --P1--G1--
P1 = S1*k;
G1: -> prot1;
```

the reaction rate of G1 will be “S1*k”.

It is also possible to modulate the inherited reaction rate. To do this, we use ellipses ('...') as shorthand for 'the formula for the element upstream of me'. Let's add a ribosome binding site that increases the rate of production of protein by a factor of three, and say that the promoter actually increases the rate of protein production by S1*k instead of setting it to S1*k:

```
dna1: --P1--RBS1--G1--
P1 = S1*k + ...;
```

```
  RBS1 = ...*3;
  G1: -> prot1;
```

Since in this model, nothing is upstream of P1, the upstream rate is set to zero, so the final reaction rate of G1 is equal to "(S1*k + 0)*3".

Valid elements of DNA strands include formulas (operators), reactions (genes), and other DNA strands. Let's wrap our model so far in a submodule, and then use the strand in a new strand:

```
model strand1()
  dna1: --P1--RBS1--G1--
  P1 = S1*k + ...;
  RBS1 = ...*3;
  G1: -> prot1;
end

model fullstrand()
  A: strand1();
  fulldna:  P2--A.dna1
  P2 = S2*k2;
end
```

In the model 'fullstrand', the reaction that produces A.prot1 is equal to "((A.S1*A.k+(S2*k2))*3)".

Operators and genes may be duplicated and appear in multiple strands:

```
  dna1:  --P1--RBS1--G1--
  dna2:  P2--dna1
  dna3:  P2--RBS2--G1
```

Strands, however, count as unique constructs, and may only appear as singletons or within a single other strand (and may not, of course, exist in a loop, being contained in a strand that it itself contains).

If the reaction rate or formula for any duplicated symbol is left at the default or if it contains ellipses explicitly ('...'), it will be equal to the sum of all reaction rates in all the strands in which it appears. If we further define our above model:

```
  dna1:  --P1--RBS1--G1--
  dna2:  P2--dna1
  dna3:  P2--RBS2--G1
  P1 = ...+0.3;
  P2 = ...+1.2;
  RBS1 = ...*0.8;
  RBS2 = ...*1.1;
  G1: -> prot1;
```

The reaction rate for the production of 'prot1' will be equal to "(((0+1.2)+0.3)*0.8) + (((0+1.2)*1.1))". If you set the reaction rate of G1 without using an ellipsis, but include it in multiple strands, its reaction rate will be a multiple of the number of strands it is a part of. For example, if you set the reaction rate

of G1 above to “k1*S1”, and include it in two strands, the net reaction rate will be “k1*S1 + k1*S1”.

The purpose of prepending or postfixing a '--' to a strand is to declare that the strand in question is designed to have DNA attached to it at that end. If exactly one DNA strand is defined with an upstream '--' in its definition in a submodule, the name of that module may be used as a proxy for that strand when creating attaching something upstream of it, and visa versa with a defined downstream '--' in its definition:

```
model twostrands
  --P1--RBS1--G1
  P2--RBS2--G2--
end

model long
  A: twostrands();
  P3--A
  A--G3
end
```

The module 'long' will have two strands: “P3--A.P1--A.RBS1--A.G1” and “A.P2--A.RBS2--A.G2--G3”.

Submodule strands intended to be used in the middle of other strands should be defined with '--' both upstream and downstream of the strand in question:

```
model oneexported
  --P1--RBS1--G1--
  P2--RBS2--G2
end

model full
  A: oneexported()
  P2--A--stop
end
```

If multiple strands are defined with upstream or downstream “--” marks, it is illegal to use the name of the module containing them as proxy.

Interactions

Some species act as activators or repressors of reactions that they do not actively participate in. Typical models do not bother mentioning this explicitly, as it will show up in the reaction rates. However, for visualization purposes and/or for cases where the reaction rates might not be known explicitly, you may declare these interactions using the same format as reactions, using different symbols instead of the “->”: for activations, use “-o”; for inhibitions, use “-|”, and for unknown interactions or for interactions which sometimes activate and sometimes inhibit, use “-(“:

```
J0: S1 + E -> SE;
i1: S2 -| J0;
i2: S3 -o J0;
i3: S4 -( J0;
```

If a reaction rate is given for the reaction in question, that reaction must include the species listed as interacting with that reaction. This, then, is legal:

```
J0: S1 + E -> SE; k1*S1*E/S2
i1: S2 -| J0;
```

because the species S2 is present in the formula "k1*S1*E/S2". If the concentration of an inhibitory species increases, it should decrease the reaction rate of the reaction it inhibits, and visa versa for activating species. The current version of libAntimony (v1.0) does not check this, but future versions may add the check.

Function Definitions

You may create user-defined functions in a similar fashion to the way you create modules, and then use these functions in Antimony equations. These functions must be basic single equations, and act in a similar manner to macro expansions. As an example, you might define the quadratic equation thus:

```
function quadratic(x, a, b, c)
  a*x^2 + b*x + c
end
```

And then use it in a later equation:

```
S3 = quadratic(s1, k1, k2, k3);
```

This would effectively define S3 to have the equation "k1*s1^2 + k2*s1 + k3".

Importing and Exporting Antimony Models

Once you have created an Antimony file, you can convert it to SBML using the 'antimony2sbml' program available from http://antimony.sourceforge.net/ This will convert each of the models defined in the Antimony text file into a separate SBML file, including the overall '__main' module. These files can then be used for simulation or visualization in other programs. This is a command-line program: the first argument is the name of the Antimony file in question, and an optional second argument will set the prefix of the SBML files to be exported:

```
antimony2sbml.exe mymodels.txt sbml/mymodels
```

will create one or more files with the prefix 'mymodels' in the 'sbml/' directory. Each file will be of the fomat "[prefix]_[modelname].xml". If no prefix is supplied, the antimony filename is used as the prefix instead, with '.txt' (if present) replaced with '_[modelname].xml'.

SBML files can also be converted to the Antimony format using 'sbml2antimony'. This works similarly to the antimony2sbml program: The first argument must be the name of the SBML file, and the second optional argument may be the name of the antimony file:

```
sbml2antimony.exe oscli.xml mymodel.txt
```

If no second argument is given '.xml' is removed from the SBML filename (if present) and '.txt' is appended.

In time, we hope that other programs will be made available to convert Antimony files to other formats. The library 'libAntimony' is available also at http://antimony.sourceforge.net/ for programmers who wish to incorporate the ability to read in Antimony files to their own programs, and an API is provided to convert the resulting models to other formats. Our own lab's TinkerCell program (http://www.tinkercell.com/) can now convert basic Antimony models to its own format, and display the result visually.