Antimony Reference

Different authoring tools have different ways of allowing the user to build models, and these approaches have individual advantages and disadvantages. In Antimony, the main approach to building models is to use a human-readable, text-based definition language, designed to interconvert between the SBML standard and a shorthand form that allows editing without the structure and overhead of working with XML directly. This guide will show you the intricacies of working with Antimony.

Table of contents

Background

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 [CellDesigner](#). 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 usable in an automated setting, such as generating models from a template metalanguage ([TemplateSB](#) is such a metalanguage for Antimony) and readable by others without translation. Antimony (so named because the chemical symbol of the element is â€˜Sbâ€™) was designed as a successor to Jarnacâ€™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 are designed to aid the creation of genetic networks specifically. Antimony is available as a library and a Python package.

Antimony is the main method of building models in [Tellurium](#), and can be used in other contexts as well. Its main features include:

- Easily define species, reactions, compartments, events, and other elements of a biological model.
- Package and re-use models as modules with defined or implied interfaces.

Change Log

The 2.14.0 release added the ability to encode algebraic rules, and added ways to add annotations and notes to objects and the model.

The 2.13.4 release changed the default SBML output to L3v2, and added basic unit names as reserved words for better import.

The 2.13.3 release removed the '@' for parsing events, and fixed '-o' interaction parsing.

The 2.13.2 release changed some maintenance features.

The 2.13.1 release added named stoichiometries, the 'rateOf' function, and instituted case senstitivity for predefined elements.

The 2.12 release added the ability to save extra 'annotation-like' elements from the 'distributions' SBML package, and fixed numerous bugs in cvterm/SBOterm setting.

The 2.11 release quashed all known memory leaks, and added the ability to define synthesis reactions with no id (i.e. ' -> S1; k1')

The 2.10 release updated support for distributions to the latest SBML release of that package, updated the default version of SBML to Level 3 version 2, added support for setting cvterms and the SBOterm,

The 2.9.1 release added the ability to convert to SBML Level 3 version 2, and SBML Level 2 version 5.

The 2.9.0 release of Antimony was largely a maintenance release: numerous bugs were fixed, and installation was streamlined, particularly for Python and conda.

The 2.8.0 release of Antimony included support for the SBML 'Flux Balance Constraints' package (version 1), as well as constraints in general.

In the 2.7.1 release of Antimony, species marked 'hasOnlySubstanceUnits=true' in SBML now have a corresponding definition in Antimony: 'species substanceOnly S1, S2'.

In the 2.7 release of Antimony, QTAntimony got several new improvements, including displayed line numbers, find/replace, and a 'go to line' option. A few new syntaxes were also added, including the ability to concisely define elements plus their assignment rules ('species S1 in C := 3+p'), and to define submodules with implied parameters ('A: mod1(1,2)').

In the 2.6 release of Antimony, some features of hierarchical translation (deletions in particular) were made more robust, and a number of built-in distribution functions were added, which are translated to SBML using the 'distributions' package, as well as using custom annotations

In the 2.5 release of Antimony, translation of Antimony concepts to and from the Hierarchical Model Composition package was developed further to be much more robust, and a new test system was added to ensure that Antimony's 'flattening' routine (which exports plain SBML) matches libSBML's flattening routine.

In the 2.4 release of Antimony, use of the Hierarchical Model Composition package constructs in the SBML translation became standard, due to the package being fully accepted by the SBML community.

In the 2.2/2.3 release of Antimony, units, conversion factors, and deletions were added.

In the 2.1 version of Antimony, the â€˜importâ€™ handling became much more robust, and it became additionally possible to export hierarchical models using the Hierarchical Model Composition package constructs for SBML level 3.

In the 2.0 version of Antimony, it became possible to export models as CellML. This requires the use of the CellML API, which is now available as an SDK. Hierarchical models are exported using CellMLâ€™s hierarchy, translated to accommodate their â€˜black boxâ€™ requirements.

Introduction and Basics

Creating a model in Antimony is designed to be very straightforward and simple. Model elements are created and defined in text, with a simple syntax.

The most common way to use Antimony is to create a reaction network, where processes are defined wherein some elements are consumed and other elements are created. Using the language of SBML, the processes are called â€˜reactionsâ€™ and the elements are called â€˜speciesâ€™, but any set of processes and elements may be modeled in this way. The syntax for defining a reaction in Antimony is to list the species being consumed, separated by a +, followed by an arrow ->, followed by another list of species being created, followed by a semicolon. If this reaction has a defined mathematical rate at which this happens, that rate can be listed next:

```
S1 -> S2; k1*S1
```

The above model defines a reaction where S1 is converted to S2 at a rate of â€˜k1*S1â€™.

This model cannot be simulated, however, because a simulator would not know what the conditions are to start the simulation. These values can be set by using an equals sign: cillator:

```
S1 -> S2; k1*S1
S1 = 10
S2 = 0
k1 = 0.1
```

The above, then, is a complete model that can be simulated by any software that understands SBML (to which Antimony models can be converted).

If you want to give your model a name, you can do that by wrapping it with the text: model [name] [reactions, etc.] end:

```
# Simple UniUni reaction with first-order mass-action kinetics
```

```
model example1
  S1 -> S2; k1*S1
  S1 = 10
  S2 = 0
  k1 = 0.1
end
```

In subsequent examples in this tutorial, we'll be using this syntax to name the examples, but for simple models, the name is optional. Later, when we discuss submodels, this will become more important.

There are many more complicated options in Antimony, but the above has enough power to define a wide variety of models, such as this oscillator:

```
model oscli
  #Reactions:
  J0:     -> S1;  J0_v0
  J1: S1 ->   ;  J1_k3*S1
  J2: S1 -> S2; (J2_k1*S1 - J2_k2*S2)*(1 + J2_c*S2^J2_q)
  J3: S2 ->   ;  J3_k2*S2

  # Species initializations:
  S1 = 0
  S2 = 1

  # Variable initializations:
  J0_v0 = 8
  J1_k3 = 0
  J2_k1 = 1
  J2_k2 = 0
  J2_c  = 1
  J2_q  = 3
  J3_k2 = 5
end
```

Model Elements

Comments

Single-line comments in Antimony can be created using the # or // symbols, and multi-line comments can be created by surrounding them with /* [comments] */.

```
/* This is an example of a multi-line
   comment for this tutorial */
model example2
  J0: S1 -> S2 + S3; k1*S1 #Mass-action kinetics
  S1 = 10  #The initial concentration of S1
  S2 = 0   #The initial concentration of S2
  S3 = 3   #The initial concentration of S3
  k1 = 0.1 #The value of the kinetic parameter from J0.
end
```

The names of the reaction and the model are saved in SBML, but any comments are not.

Reactions

Reactions can be created with multiple reactants and/or products, and the stoichiometries can be set by adding a number before the name of the species:

```
# Production of S1
    -> S1;                    k0
# Conversion from S1 to S2
S1 -> S2;                 k1*S1
# S3 is the adduct of S1 and S2
S1 + S2 -> S3;          k2*S1*S2
# Dimerization of S1
2 S1 -> S2;             k3*S1*S1
# More complex stoichiometry
S1 + 2 S2 -> 3 S3 + 5 S4; k4*S1*S2*S2
# Degradation of S4
S4 -> ; k5*S4
```

Rate Laws and Initializing Values

Reactions can be defined with a wide variety of rate laws

```
model pathway()
  # Examples of different rate laws and initialization

  S1 -> S2; k1*S1
  S2 -> S3; k2*S2 - k3*S3
  S3 -> S4; Vm*S3/(Km + S3)
  S4 -> S5; Vm*S4^n/(Km + S4)^n

  S1 = 10
  S2 = 0
  S3 = 0
  S4 = 0
  S5 = 0
  k1 = 0.1
  k2 = 0.2
  k3 = 0.2
  Vm = 6.7
  Km = 1E-3
  n = 4
end
```

Defining parameters, species, and compartments.

By default, any named element in an Antimony model is translated as an SBML â€˜parameterâ€™. If it is used in a reaction, it is translated as a species. If one element is â€˜inâ€™ a second element, the second element becomes a compartment.

However, it is possible to define any element as a species or compartment directly when they do not appear in those contexts:

```
p = 3
species s = 4
compartment c = 5
```

If, for example, a species is changing due to a rate rule and does not appear in a reaction, you must declare it to be a species:

```
species S1 = 1.3
S1' = 0.4
```

Boundary Species

Boundary species are those species which are unaffected by the model. Usually this means they are fixed. There are two ways to declare boundary species.

1. Using a dollar sign to indicate that a particular species is fixed:

```
model pathway()
  # Example of using $ to fix species

  $S1 ->  S2; k1*S1
  S2 ->  S3; k2*S2
  S3 -> $S4; k3*S3
end
```

2. Using the const keyword to declare species are fixed:

```
model pathway()
  # Examples of using the const keyword to fix species

  const S1, S4
  S1 -> S2; k1*S1
  S2 -> S3; k2*S2
  S3 -> S4; k3*S3
end
```

Substance-only Species

When a species ID shows up in math, it usually means 'the concentration of the species'. If instead it should be 'the amount of the species', you can indicate this by using the 'substanceOnly' keyword:

```
  substanceOnly species S1 in C
  S1 -> S2; k1*S1/C
```

In the produced SBML, these species will be flagged hasOnlySubstanceUnits="true"

Compartments

For multi-compartment models, or models where the compartment size changes over time, you can define the compartments in Antimony by using the `compartment` keyword, and designate species as being in particular compartments with the `in` keyword:

```
model pathway()
  # Examples of different compartments

  compartment cytoplasm = 1.5, mitochondria = 2.6
  const S1 in mitochondria
  var S2 in cytoplasm
  var S3 in cytoplasm
  const S4 in cytoplasm

  S1 -> S2; k1*S1
  S2 -> S3; k2*S2
  S3 -> S4; k3*S3
end
```

Assignments

You can also initialize elements with more complicated formulas than simple numbers:

```
model pathway()
  # Examples of different assignments

  A = 1.2
  k1 = 2.3 + A
  k2 = sin(0.5)
  k3 = k2/k1

  S1 -> S2; k1*S1
  S2 -> S3; k2*S2
  S3 -> S4; k3*S3
end
```

Assignments in Time

If you want to define some elements as changing in time, you can either define the formula a variable equals at all points in time with a `:=`, or you can define how a variable changes in time using `x'` (a [rate rule](#)) in which case you€™ll also need to define its initial starting value. The keyword `time` represents time.

```
model pathway()
  # Examples of assignments that change in time

  k1 := sin(time)   #  k1 will always equal the sine of time
  k2  = 0.2
  k2' = k1          #' k2 starts at 0.2, and changes according to the value
                    #   of k1: d(k2)/dt = k1
```

```
  S1 -> S2; k1*S1
  S2 -> S3; k2*S2
end
```

Piecewise Assignments

You can use `piecewise` to define piecewise assignments.

```
model pathway()
  # Examples of piecewise assignments
  $Xo -> S1; k1*Xo;
  S1 -> S2; k2*S1;
  S2 -> $X1; k3*S2;

  k1 := piecewise(0.1, time > 50, 20)

  k2 = 0.45; k3 = 0.34; Xo = 5;
end
```

Above will return `k1 = 0.1` if `time > 50` and `20` otherwise. A more complicated piecewise assignment can be defined as well.

```
model pathway()
  $Xo -> S1; k1*Xo;
  S1 -> S2; k2*S1;
  S2 -> $X1; k3*S2;

  k1 := piecewise(5, time > 20, 8, S2 < 100, 15)

  k2 = 0.45; k3 = 0.34; Xo = 5;
end
```

The above piecewise call will return 5 if time > 20, else it will return 8 if S2 < 100, else it will return 15. The piecewise function has this general "do this if this is true, else …" pattern, and can be extended to include as may conditions as needed.

Events (basic)

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

```
at (x>5): y=3, x=r+2
```

In a model with this event, at any moment when x transitions from being less than or equal to 5 to being greater to five, y will be assigned the value of 3, and x will be assigned the value of r+2, using whatever value r has at that moment. The following model sees the conversion of S1 to S2 until a threshold is reached, at which point the cycle is reset.

```
model reset()
```

```
  S1 -> S2; k1*S1

  E1: at (S2>9): S2=0, S1=10

  S1 = 10
  S2 = 0
  k1 = 0.5
end
```

For more advanced usage of events, see [below](#).

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 and use it in a later equation as follows:

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

model quad1
  S3 := quadratic(s1, k1, k2, k3);
end
```

This effectively defines S3 to always equal the equation `k1*s1^2 + k2*s1 + k3`.

Annotation

Antimony elements can be annotated with URNs using annotation keywords You can see the [full list](#) below, but in general, you annotate in the following way:

```
//Species
species Glcin in comp, MgATP in comp

// CV terms:
comp  identity "http://identifiers.org/go/GO:0005737"
Glcin identity "http://identifiers.org/chebi/CHEBI:17234",
               "http://identifiers.org/kegg.compound/C00293"
MgATP part "http://identifiers.org/chebi/CHEBI:25107",
           "http://identifiers.org/chebi/CHEBI:15422"
```

Any Antimony element with an id may be annotated in this way, including the model itself. Inside a model definition, the model itself may be annotated using the 'model' keyword:

```
  model foo()
    model model_entity_is
"http://identifiers.org/biomodels.db/BIOMD0000000004"
    model description "http://identifiers.org/pubmed/1833774"
    model origin "http://identifiers.org/biomodels.db/BIOMD0000000003"
```

```
    model taxon "http://identifiers.org/taxonomy/8292"
    model created "2005-02-08T17:34:02Z"
    model modified "2012-12-11T15:30:15Z"
  end
```
You can also define an element's 'notes', using the 'notes' keyword. If the notes take more than one line, you can group them together using three tick marks '\`\`\`':
```
  model notes ```
      <p>This model represents the inactive forms of CDC-2 Kinase and Cyclin
      Protease as separate species, unlike the ODEs in the published paper,
in
      which the equations for the inactive forms are substituted into the
      equations for the active forms using a mass conservation rule
      M+MI=1,X+XI=1. Mass is still conserved in this model through the
      explicit reactions M<->MI and X<->XI. The terms in the
      kinetic laws are identical to the corresponding terms in the kinetic
      laws in the published paper.</p>
```
```

## Modular Models

Antimony was actually originally designed to allow the modular creation of models, and has a basic syntax set up to do so. For a full discussion of Antimony modularity, see below, but at the most basic level, you define a re-usable module with the 'model' syntax, followed by parentheses where you define the elements you wish to expose, then import it by using the model's name, and the local variables you want to connect to that module

```
# This creates a model 'side_reaction', exposing the variables 'S' and 'k1':
model side_reaction(S, k1)
  J0: S + E -> SE; k1*k2*S*E - k2*ES;
  E = 3;
  SE = E+S;
  k2 = 0.4;
end

# In this model, 'side_reaction' is imported twice:
model full_pathway
    -> S1; k1
  S1 -> S2; k2*S1
  S2 ->   ; k3*S2

  A: side_reaction(S1, k4)
  B: side_reaction(S2, k5)

  S1 = 0
  S2 = 0
  k1 = 0.3
  k2 = 2.3
  k3 = 3.5
  k4 = 0.0004
  k5 = 1

end
```

In this model, `A` is a submodel that creates a side-reaction of `S1` with `A.E` and `A.SE`, and `B` is a submodel that creates a side-reaction of `S2` with `B.E` and `B.SE`. It is important to note that there is no connection between `A.E` and `B.E` (nor `A.ES` and `B.ES`): they are completely different species in the model.

Importing 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.

Signals

Signals can be generated by combining assignment rules with events.

Step Input

The simplest signal is input step. The following code implements a step that occurs at time = 20 with a magnitude of f. A trigger is used to set a trigger variable alpha which is used to initate the step input in an assignment expression.

```
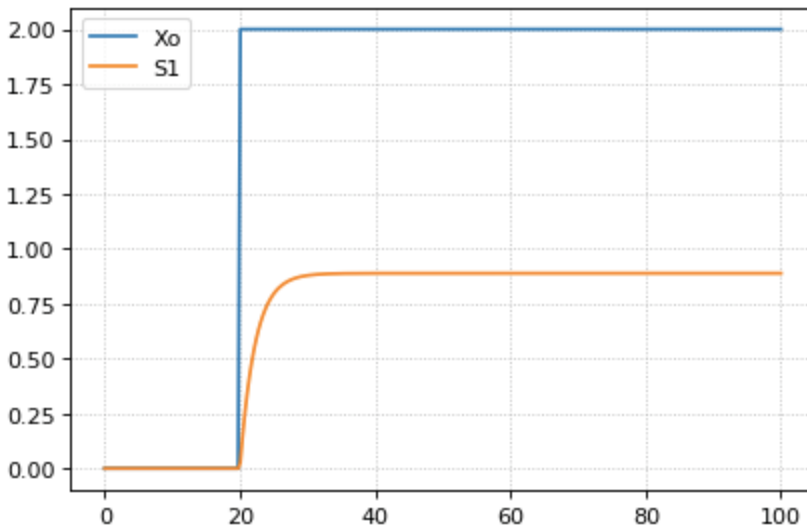$Xo -> S1; k1*Xo;
S1 -> $X1; k2*S1;

k1 = 0.2; k2 = 0.45;

alpha = 0; f = 2
Xo := alpha*f
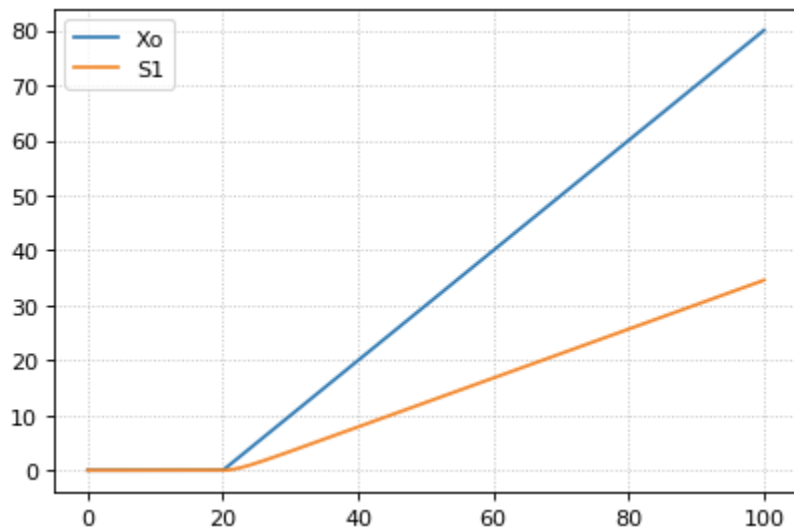at time > 20:
    alpha = 1
```

Step Input

Ramp

The following code starts a ramp at 20 time units by setting the p1 variable to one. This variable is used to acticate a ramp function.

```
$Xo -> S1;  k1*Xo;
S1 -> $X1;  k2*S1;

k1 = 0.2;  k2 = 0.45;

p1 = 0;
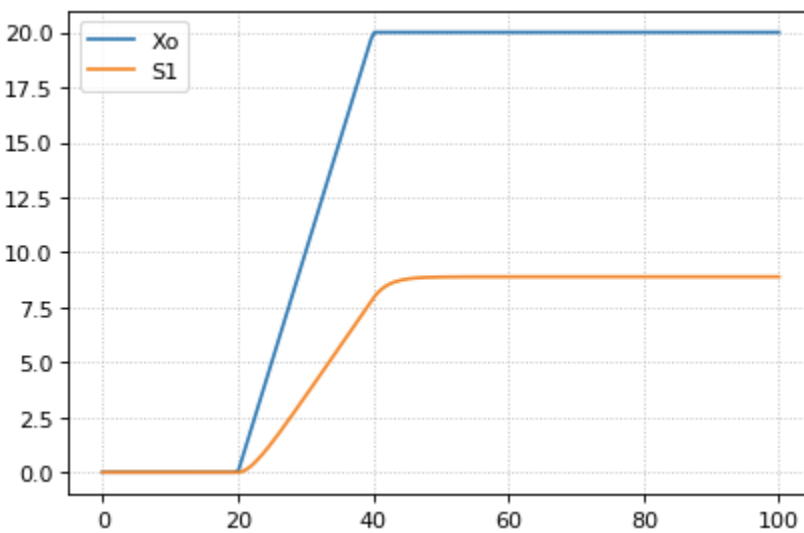Xo := p1*(time - 20)
at time > 20:
    p1 = 1
```


Ramp

Ramp then Stop

The following code starts a ramp at 20 time units by setting the p1 variable to one and then stopping the ramp 20 time units later. At 20 time units later a new term is switched on which subtract the ramp slope that results in a horizontal line.

```
$Xo -> S1; k1*Xo;
S1 -> $X1; k2*S1;

k1 = 0.2; k2 = 0.45;

p1 = 0; p2 = 0
Xo := p1*(time - 20) - p2*(time - 40)
at time > 20:
    p1 = 1
at time > 40:
    p2 = 1
```


Ramp Then Stop

Pulse

The following code starts a pulse at 20 time units by setting the p1 variable to one and then stops the pulse 20 time units later by setting p2 equal to zero.

```
$Xo -> S1; k1*Xo;
S1 -> $X1; k2*S1;

k1 = 0.2; k2 = 0.45;

p1 = 0; p2 = 1
Xo := p1*p2
at time > 20:
    p1 = 1
at time > 40:
    p2 = 0
```

Pulse

Sinusoidal Input

The following code starts a sinusoidal input at 20 time units by setting the p1 variable to one.

```
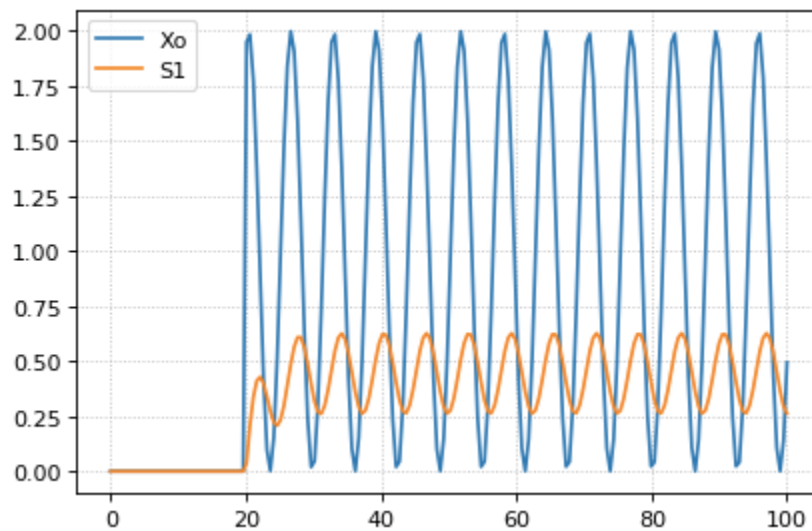$Xo -> S1;  k1*Xo;
S1 -> $X1;  k2*S1;
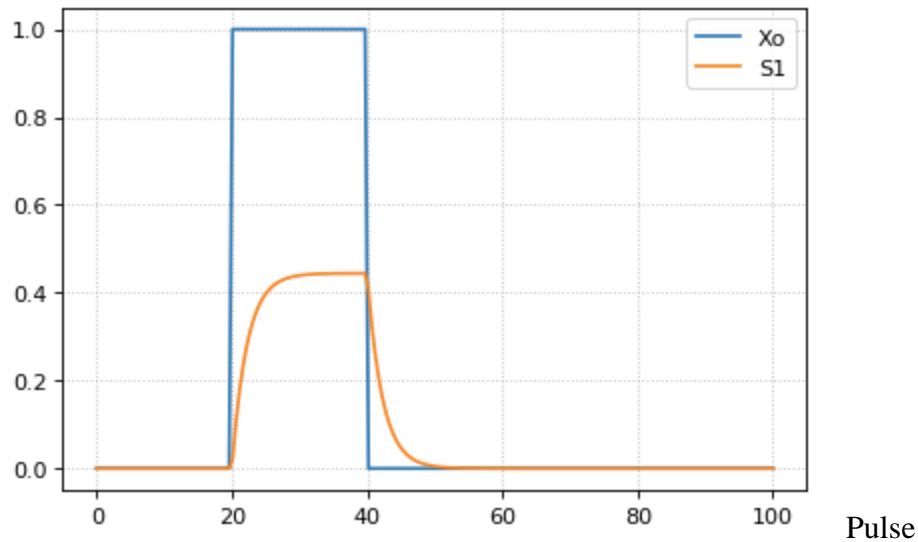
k1 = 0.2;  k2 = 0.45;

p1 = 0;
Xo := p1*(sin (time) + 1)
at time > 20:
    p1 = 1
```


Sinusoidal Input

Language Reference

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.

If you want, you can define a reaction to be irreversible by using `=>` instead of `->`:

```
J0: S1 + E => ES;
```

However, if you additionally provide a reaction rate, that rate is not checked to ensure that it is compatible with an irreversible reaction.

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 â€˜undefinedâ€™; â€˜undefinedâ€™ symbols may later be declared or used as species or as â€˜formulasâ€™, Antimonyâ€™s term for constants and packaged equations like SBMLâ€™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 this is not required.

Substance-only species

In SBML, species can be defined such that when their ID is used in math, that symbol means either 'the concentration of the species' or 'the amount of the species'. In Antimony, by default, a species symbol is defined as meaning its concentration. However, it can be defined to mean the amount by using the 'substanceOnly' keyword:

substanceOnly species S1;

Now, whenever 'S1' is used in the model, it is a reference to the species amount, and not its concentration. Defining an initial amount is still accomplished with the same syntax:

S1 = 2.5/C1;

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:

```
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, prepend 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, prepended 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.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.

As a final note: use of the `is` keyword is not restricted to elements inside submodules. As a result, if you wish to change the name of an element (if, for example, you want the reactions to look simpler in Antimony, but wish to have a more descriptive name in the exported SBML), you may use `is` as well:

```
A -> B;
A is ABA;
B is ABA8OH;
```

is equivalent to writing:

```
ABA -> ABA8OH;
```

Module conversion factors

Occasionally, the unit system of a submodel will not match the unit system of the containing model, for one or more model elements. In this case, you can use conversion factor constructs to bring the submodule in line with the containing model.

If time is different in the submodel (affecting reactions, rate rules, delay, and 'time'), use the `timeconv` keyword when declaring the submodel:

```
A1: submodel(), timeconv=60;
```

This construct means that one unit of time in the submodel multiplied by the time conversion factor should equal one unit of time in the parent model.

Reaction extent may also be different in the submodel when compared to the parent model, and may be converted with the `extentconv` keyword:

```
A1: submodel(), extentconv=1000;
```

This construct means that one unit of reaction extent in the submodel multiplied by the extent conversion factor should equal one unit of reaction extent in the parent model.

Both time and extent conversion factors may be numbers (as above) or they may be references to constant parameters. They may also both be used at once:

```
A1: submodel(), timeconv=tconv, extentconv=xconv;
```

Individual components of submodels may also be given conversion factors, when the `is` keyword is used. The following two constructs are equivalent ways of applying conversion factor `cf` to the synchronized variables `x` and `A1.y`:

```
A1.y * cf is x;
A1.y is x / cf;
```

When flattened, all of these conversion factors will be incorporated into the mathematics.

Submodel deletions

Sometimes, an element of a submodel has to be removed entirely for the model to make sense as a whole. A degradation reaction might need to be removed, for example, or a now-superfluous species. To delete an element of a submodel, use the `delete` keyword:

```
delete A1.S1;
```

In this case, `S1` will be removed from submodel `A1`, as will any reactions `S1` participated in, plus any mathematical formulas that had `S1` in them.

Similarly, sometimes it is necessary to clear assignments and rules to a variable. To accomplish this, simply declare a new assignment or rule for the variable, but leave it blank:

```
A1.S1  = ;
A1.S2 := ;
A1.S3' = ;
```

This will remove the appropriate initial assignment, assignment rule, or rate rule (respectively) from the submodel.

Constant and variable symbols

Some models have 'boundary species' in their reactions, or species whose concentrations do not change as a result of participating in a reaction. To declare that a species is a boundary species, use the 'const' keyword:

```
const S1;
```

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

```
const species S1;
```

If a symbol appears as a participant in a reaction, Antimony will recognize that it is a species automatically, so the use of the keyword 'species' is not required. If, however, you have a species which never appears in a reaction, you will need to use the 'species' keyword.

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;
```

Alternatively, you may declare a species to be a boundary species by prepending a '$' in front of it:

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

This would set the level of 'E' to be constant. You can use this symbol in declaration lists as well:

```
species S1, $S2, $S3, S4, S5, $S6;
```

This declares six species, three of which are variable (by default) and three of which are constant.

Likewise, formulas are constant by default. They may be initialized with an equals sign, with either a simple or a complex formula:

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

You may also explicitly declare whether they are constant or variable:

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

and be more specific and declare that both are formulas:

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

Variables defined with an equals sign are assigned those values at the start of the simulation. In SBML terms, they use the 'Initial Assignment' values. If the formula is to vary during the course of the simulation, use the Assignment Rule (or Rate Rule) syntax, described later.

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', 'k2', and even the formula '-log10(pH)' 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);
```

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*comp1;
```

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 at the moment when 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: variable1=formula1, variable2=formula2 [etc];
```

such as:

```
at (x>5): y=3, x=r+2;
```

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

```
E1: at(x>=5): y=3, x=r+2;
```

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

In addition, there are a number of concepts in SBML events that can now be encoded in
Antimony. If event assignments are to occur after a delay, this can be encoded by using the
'after' keyword:

```
E1: at 2 after (x>5): y=3, x=r+2;
```

This means to wait two time units after x transitions from less than five to more than five, then
change y to 3 and x to r+2. The delay may also itself be a formula:

```
E1: at 2*z/y after (x>5): y=3, x=r+2;
```

For delayed events (and to a certain extent with simultaneous events, discussed below), one
needs to know what values to use when performing event assignments: the values from the time
the event was triggered, or the values from the time the event assignments are being executed?
By default (in Antimony, as in SBML Level 2) the first holds true: event assignments are to use
values from the moment the event is triggered. To change this, the keyword 'fromTrigger'
is used:

```
E1: at 2*z/y after (x>5), fromTrigger=false: y=3, x=r+2;
```

You may also declare 'fromTrigger=true' to explicitly declare what is the default.

New complications can arise when event assignments from multiple events are to execute at the
same time: which event assignments are to be executed first? By default, there is no defined
answer to this question: as long as both sets of assignments are executed, either may be executed
first. However, if the model depends on a particular order of execution, events may be given
priorities, using the priority keyword:

```
E1: at ((x>5) && (z>4)), priority=1: y=3, x=r+2;
E2: at ((x>5) && (q>7)), priority=0: y=5: x=r+6;
```

In situations where z>4, q>7, and x>5, and then x increases, both E1 and E2 will trigger at the same time. Since both modify the same values, it makes a difference in which order they are executed-in this case, whichever happens last takes precedence. By giving the events priorities (higher priorities execute first) the result of this situation is deterministic: E2 will execute last, and y will equal 5 and not 3.

Another question is whether, if at the beginning of the simulation the trigger condition is â€~trueâ€™, it should be considered to have just transitioned to being true or not. The default is no, meaning that no event may trigger at time 0. You may override this default by using the â€~t0â€™ keyword:

```
E1: at (x>5), t0=false: y=3, x=r+2;
```

In this situation, the value at t0 is considered to be false, meaning it can immediately transition to true if x is greater than 5, triggering the event. You may explicitly state the default by using â€~t0 = trueâ€™.

Finally, a different class of events is often modeled in some situations where the trigger condition must persist in being true from the entire time between when the event is triggered to when it is executed. By default, this is not the case for Antimony events, and, once triggered, all events will execute. To change the class of your event, use the keyword â€~persistentâ€™:

```
E1: at 3 after (x>5), persistent=true: y=3, x=r+2;
```

For this model, x must be greater than 5 for three seconds before executing its event assignments: if x dips below 5 during that time, the event will not fire. To explicitly declare the default situation, use â€~persistent=falseâ€™.

The ability to change the default priority, t0, and persistent characteristics of events was introduced in SBML Level 3, so if you translate your model to SBML Level 2, it will lose the ability to define functionality other than the default. For more details about the interpretation of these event classifications, see the SBML Level 3 specification.

Assignment Rules

In some models, species and/or variables change in a manner not described by a reaction. When a variable receives a new value at every point in the model, this can be expressed in an assignment rule, which in Antimony is formulated with a â€~:=â€™ as:

```
Ptot := P1 + P2 + PE;
```

In this example, â€~Ptotâ€™ will continually be updated to reflect the total amount of â€~Pâ€™ present in the model.

Each symbol (species or formula) may have only one assignment rule associated with it. If an Antimony file defines more than one rule, only the last will be saved.

When species are used as the target of an assignment rule, they are defined to be ‘boundary species’ and thus ‘const’. Antimony doesn't have a separate syntax for boundary species whose concentrations never change vs. boundary species whose concentrations change due to assignment rules (or rate rules, below). SBML distinguishes between boundary species that may change and boundary species that may not, but in Antimony, all boundary species may change as the result of being in an Assignment Rule or Rate Rule.

Rate Rules

Rate rules define the change in a symbol's value over time instead of defining its new value. In this sense, they are similar to reaction rate kinetics, but without an explicit stoichiometry of change. These may be modeled in Antimony by appending an apostrophe to the name of the symbol, and using an equals sign to define the rate:

```
S1' =  V1*(1 - S1)/(K1 + (1 - S1)) - V2*S1/(K2 + S1)
```

Note that unlike initializations and assignment rules, formulas in rate rules may be self-referential, either directly or indirectly.

Any symbol may have only one rate rule or assignment rule associated with it. Should it find more than one, only the last will be saved.

Algebraic Rules

Algebraic rules are defined as equations that are always true, but do not declare which variable or variables should be adjusted to ensure that they are true. Algebraic rules have somewhat limited support in some simulators, and are not supported by Tellurium, so they are provide by Antimony solely for users who wish to export the model and use it in other systems.

To declare an algebraic rule, optionally give it a name, and then declare "0 = [formula]":

```
  0 =  S1*k1 - 10
  alg2: 0 = S2*k2 - 20
```

Display Names

When some tools visualize models, they make a distinction between the ‘id’ of an element, which must be unique to the model and which must conform to certain naming conventions, and the ‘name’ of an element, which does not have to be unique and which has much less stringent naming requirements. In Antimony, it is the id of elements which is used everywhere. However, you may also set the ‘display name’ of an element by using the ‘is’ keyword and putting the name in quotes:

```
A.k1 is "reaction rate k1";
S34  is "Ethyl Alcohol";
```

Units

While units do not affect the mathematics of SBML or Antimony models, you can
define them in Antimony for annotation purposes by using the unit keyword:

```
unit substance = 1e-6 mole;
unit hour = 3600 seconds;
```

Adding an 'â€˜sâ€™' to the end of a unit name to make it plural is fine when
defining a unit: 3600 second is the same as 3600 seconds. Compound units may
be created by using formulas with *, /, and ^. However, you must use base
units when doing so ('â€˜base unitsâ€™' defined as those listed in Table 2 of
the SBML Level 3 Version 1 specification, which mostly are SI and SI-derived
units).

```
unit micromole = 10e-6 mole / liter;
unit daily_feeding = 1 item / 86400 seconds
unit voltage = 1000 grams * meters^2 / seconds^-3 * ampere^-1
```

You may use units when defining formulas using the same syntax as above: any
number may be given a unit by writing the name of the unit after the number.
When defining a symbol (of any numerical type: species, parameter,
compartment, etc.), you can either use the same technique to give it an
initial value and a unit, or you may just define its units by using the
'â€˜hasâ€™' keyword:

```
unit foo = 100 mole/5 liter;
x = 40 foo/3 seconds; # '40' now has units of 'foo' and '3' units of
'seconds'.
y = 3.3 foo;          # 'y' is given units of 'foo' and an initial
                      #   value of '3.3'.
z has foo;            # 'z' is given units of 'foo'.
```

Antimony does not calculate any derived units: in the above example, 'â€˜xâ€™'
is fully defined in terms of moles per liter per second, but it is not
annotated as such.

As with many things in Antimony, you may use a unit before defining it: 'â€˜x
= 10 mlâ€™' will create a parameter x and a unit 'â€˜mlâ€™'.

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 ->: 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 vice versa for activating species. The current version of libAntimony (v2.4) does not check this, but future versions may add the check.

When the reaction rate is not known, species from interactions will be added to the SBML 'listOfModifiers' for the reaction in question. Normally, the kinetic law is parsed by libAntimony and any species there are added to the list of modifiers automatically, but if there is no kinetic law to parse, this is how to add species to that list.

Predefined Function Definitions

In addition to the user-defined function definitions, there are several built-in functions defined in Antimony. All of the functions present in the MathML subset used in SBML Level 3 Level 2 are likewise defined here, and include:

abs, and, arccos, arccosh, arccot, arccoth, arccsc, arccsch, arcsec, arcsech, arcsin,
arcsinh, arctan, arctanh, ceiling, cos, cosh, cot, coth, csc, csch, divide, eq, exp,
factorial, floor, geq, gt, leq, ln, log, lt, minus, neq, not, or, piecewise, plus,
power, root, sec, sech, sin, sinh, tan, tanh, times, and xor.

In addition, the constants

true, false, notanumber, pi, avogadro, infinity, and exponentiale

are all allowed.

As of Antimony v2.12, the following distributions are also allowed, and will
be added to the translated SBML file if used:

normal(mean, stddev),
normal(mean, stddev, min, max),
uniform(min, max),
bernoulli(prob),
binomial(nTrials, probabilityOfSuccess),
binomial(nTrials, probabilityOfSuccess, min, max),
cauchy(location, scale),
cauchy(location, scale, min, max),
chisquare(degreesOfFreedom),
chisquare(degreesOfFreedom, min, max),
exponential(rate),
exponential(rate, min, max),
gamma(shape, scale),
gamma(shape, scale, min, max),
laplace(location, scale),
laplace(location, scale, min, max),
lognormal(mean, stdev),
lognormal(mean, stdev, min, max),
poisson(rate),
poisson(rate, min, max),
rayleigh(scale), and
rayleigh(scale, min, max).

The â€˜truncatedâ€™ forms of all functions allow one to define inclusive
boundaries, meaning that the returned value must fall between the min and the
max values given.

Uncertainty Information

The SBML 'Distributions' package introduced a variety of ways to store information about the uncertainty of model elements. Antimony is now extended to also store this same information, through the following syntax:

```
A.mean = x
A.stdev = x   (or A.standardDeviation = x)
A.coefficientOfVariation = x
A.kurtosis = x
A.median = x
A.mode = x
A.sampleSize = x
A.skewness = x
A.standardError = x
A.variance = x
A.confidenceInterval = {x, y}
A.credibleInterval = {x, y}
A.interquartileRange = {x,y}
A.range = {x,y}
A.distribution = function()
A.distribution is "http://uri"
A.externalParameter = x || {x,y} || function()
A.externalParameter is "http://uri"
```

Where A may be any symbol in Antimony with mathematical meaning; x and y may both be either a symbol or a value (i.e. A.mean=2.4; A.confidenceInterval={S1, 8.2}); function() may be any mathematical formula; and "http://uri" is a URI that defines the given distribution or externalParameter.

Annotation Keywords

Antimony model elements may also be annotated with their SBO terms and cvterms, using the following syntax:

```
A.sboTerm = 236 or A.sboTerm = SBO:00000236
A identity "cvterm" or A biological_entity_is "cvterm"
A hasPart "cvterm" or A part "cvterm"
A isPartOf "cvterm" or A parthood "cvterm"
A isVersionOf "cvterm" or A hypernym "cvterm"
A hasVersion "cvterm" or A version "cvterm"
A isHomologTo "cvterm" or A homolog "cvterm"
A isDescribedBy "cvterm" or A description "cvterm"
A isEncodedBy "cvterm" or A encoder "cvterm"
```

```
A encodes "cvterm" or A encodement "cvterm"
A occursIn "cvterm" or A container "cvterm"
A hasProperty "cvterm" or A property "cvterm"
A isPropertyOf "cvterm" or A propertyBearer "cvterm"
A hasTaxon "cvterm" or A taxon "cvterm"
A created "YYYY-MM-DDThh:mm:ssTZD" where TZD is either Z or +/- HH:MM
A modified "YYYY-MM-DDThh:mm:ssTZD" where TZD is either Z or +/- HH:MM
A creator "creator"
A creator.name "full name"
A creator.givenName "given name"
A creator.familyName "family name"
A creator.organization "organization"
A creator.email "email address"
A notes "notes"
```

Where A is any model ID or the word 'model' for the model itself, and cvterm
is a URI like "http://identifiers.org/uniprot/P12999".   If

there are multiple creators, or multiple modification times, you can

distinguish between them by adding a number:

```
A creator1.name "Hugh Barrett"
A creator2.name "Nancy Smalls"
A modified1 "2012-12-11T15:30:15Z"
A modified2 "2013-01-15T12:25:55Z"
```

You can also set the individual components of the 'created' or 'modified'

date by keyword:

```
A created.year "YYYY"
A created.month "MM"
A created.day "DD"
A created.hour "hh"
A created.minute "mm"
A created.second "ss"
A created.time "hh:mm:ss"
```

Flux Balance Constraints

In some models, reaction rates are not known specifically, but one can place certain constraints on those reactions, and then apply an objective function (such as 'maximize growth') to try to discern a likely set of reaction rates. In SBML, the package that lets you define these constraints and objective functions is known as the 'Flux Balance Constraints' package. As of v2.8.0 of Antimony, these constraints can now be defined in Antimony as well, using equalities and inequalities <, >, <=, >=, and ==. If we assume that all J variables are reactions, the following definitions are all Flux Balance constraints:

```
0 <= J0
J1 <= 1000
-10 <= J2 <= 10
```

Constraints that do not involve the ID of a reaction by itself will be translated as core SBML constraints. (Flux Balance constraints are also translated as core constraints, for consistency.)

The objective function is defined using either the keyword maximize or minimize. It may be named by prepending the statement with that name, followed by a colon:

```
maximize J1
obj1: minimize J2
```

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. Most SBML files will not contribute to this module, unless the name of the model in the file is __main (for example, if it was created by the antimony converter).

By default, libantimony will examine the 'import' text to determine whether it is a relative or absolute filename, and, if relative, will prepend the directory of the working file to the import text before attempting to load the file. If it cannot find it there, it is possible to tell the libantimony API to look in different directories for files loaded from import statements.

However, if the working directory contains a .antimony file, or if one of the named directories contains a .antimony file, import statements can be subverted. Each line of this file must contain three tab-delimited strings: the name of the file which contains an import statement, the text of the import statement, and the filename where the program should look for the file. Thus, if a file file1.txt contains the line import "file2.txt", and a .antimony file is discovered with the line:

file1.txt file2.txt   antimony/import/file2.txt

The library will attempt to load 'antimony/import/file2.txt' instead of looking for 'file2.txt' directly. For creating files in-memory or when reading antimony models from strings, the first string may be left out:

file2.txt antimony/import/file2.txt

The first and third entries may be relative filenames: the directory of the .antimony file itself will be added internally when determining the file's actual location. The second entry must be exactly as it appears in the first file's 'import' directive, between the quotation marks.

Importing and Exporting Antimony Models

Once you have created an Antimony file, you can convert it to SBML or CellML using 'sbtranslate' or the 'QTAntimony' visual editor (both available from http://antimony.sourceforge.net/) This will convert each of the models defined in the Antimony text file into a separate SBML model, including the overall '__main' module (if it contains anything). These files can then be used for simulation or visualization in other programs.

QTAntimony can be used to edit and translate Antimony, SBML, and CellML models. Any file in those three formats can be opened, and from the 'View' menu, you can turn on or off the SBML and CellML tabs. Select the tabs to translate and view the working model in those different formats.

The SBML tabs can additionally be configured to use the 'Hierarchical Model Composition' package constructs. Select 'Edit/Flatten SBML tab(s)' or hit control-F to toggle between this version and the old 'flattened' version of SBML. (To enable this feature if you compile Antimony yourself, you will need the latest versions of libSBML with the SBML 'comp' package enabled, and to select 'WITH_COMP_SBML' from the CMake menu.)

As there were now several different file formats available for translation, the old command-line translators still exist (antimony2sbml; sbml2antimony), but have been supplanted by the new 'sbtranslate' executable. Instructions for use are available by running sbtranslate from the command line, but in brief: any number of files to translate may be added to the command line, and the desired output format is given with the '-o' flag: -o antimony, -o sbml, -o cellml, or -o sbml-comp (the last to output files with the SBML 'comp' package constructs).

Examples:

sourceCode bash sbtranslate model1.txt model2.txt -o sbml

will create one flattened SBML file for the main model in the two Antimony files in the working directory. Each file will be of the format '[prefix].xml', where [prefix] is the original filename with '.txt' removed (if present).

sourceCode bash sbtranslate oscli.xml ffn.xml -o antimony

will output two files in the working directory: 'oscli.txt' and 'ffn.txt' (in the antimony format).

```
sourceCode bash sbtranslate model1.txt -o sbml-comp
```

will output 'model1.xml' in the working directory, containing all models in the 'model1.txt' file, using the SBML 'comp' package.

Appendix: Converting between SBML and Antimony

For reference, here are some of the differences you will see when converting models between SBML and Antimony:

- Local parameters in SBML reactions become global parameters in Antimony, with the reaction name prepended. If a different symbol already has the new name, a number is appended to the variable name so it will be unique. These do not get converted back to local parameters when converting Antimony back to SBML.

- Algebraic rules in SBML disappear in Antimony.

- Any element with both a value (or an initial amount/concentration for species) and an initial assignment in SBML will have only the initial assignment in Antimony.

- Stoichiometry math in SBML disappears in Antimony.

- All constant=true species in SBML are set const in Antimony, even if that same species is set boundary=false.

- All boundary=true species in SBML are set const in Antimony, even if that same species is set constant=false.

- Boundary ('const') species in Antimony are set boundary=true and constant=false in SBML.

- Variable ('var') species in Antimony are set boundary=false and constant=false in SBML.

- Modules in Antimony are flattened in SBML (unless you use the comp option).

- DNA strands in Antimony disappear in SBML.

- DNA elements in Antimony no longer retain the ellipses syntax in SBML, but the effective reaction rates and assignment rules should be accurate, even for elements appearing in multiple DNA strands. These reaction rates and assignment rules will be the sum of the rate at all duplicate elements within the DNA strands.

- Any symbol with the MathML csymbol 'time' in SBML becomes 'time' in Antimony.

- Any formula with the symbol 'time' in it in Antimony will become the MathML csymbol 'time' in in SBML.

- The MathML csymbol 'delay' in SBML disappears in Antimony.

- Any SBML version 2 level 1 function with the MathML csymbol 'time' in it will become a local variable with the name 'time_ref' in Antimony. This 'time_ref' is added to the function's interface (as the last in the list of symbols), and any uses of the function are modified to use 'time' in the call. In other words, a function 'function(x, y): x+y*time' becomes ''function(x, y, time_ref): x
  -
  -
  o y*time_ref', and formulas that use 'function(A, B)' become 'function(A, B, time)'
  -

- A variety of Antimony keywords, if found in SBML models as IDs, are renamed to add an appended '_'. So the ID compartment becomes compartment_, model becomes model_, etc.

Further Reading

- Lucian Smith's [example models](#) show how to use the [comp package](#).

- [This manual](#) in PDF format.