Introduction to YNelson and YPetri (hands-on tutorial)

June 24, 2016

Introduction

This document is a hands-on guide to `YPetri`, `YNelson` and *Nelson nets*, which in turn rely on Ruby language. It is not assumed you are familiar with Ruby, you can follow this guide even if you never heard about it (but you will have to install it). You more than likely never heard about Nelson nets. Nelson net is a novel concept of *Petri nets* crossed with Ted Nelson's *ZZ structures*. Do not worry about them: just follow the guide and you will learn everything in time. The text of this guide lets you know when new keywords and are introduced. When mentioned for the first time, Ruby keywords and terms are highlighted in red, Petri net terms in green, and `YPetri / YNelson` keywords and terms in blue throughout this document.

`YNelson` serves to specify and simulate Nelson nets, a derivative of *functional Petri nets* that live in *ZZ space*. (You don't need to worry about ZZ space yet.) `YPetri` serves to specify and simulate Petri nets. Petri nets are one of the basic formalisms in systems simulation. I designed `YPetri` and `YNelson` because I needed to specify biochemical systems and I believed a Ruby DSL would be ideal for that task. Use of Petri nets for specifying biochemical systems is reviewed eg. in Bos [2008] and Koch [2015]. `YPetri` provides a universal Petri net abstraction similar, but not identical with *hybrid functional Petri net (HFPN)* introduced by Matsuno et al. [2011]. `YPetri` / `YNelson` serve equally well not just for biochemical systems, but for arbitrary dynamical systems.

Finally, please forgive me for taking your time with this guide. If you want to seriously work with *complicated* Petri Nets, no tool can avoid taking you through the process of learning the language and the tool's interface. Visual Petri net modelling tools are nice for simple tasks, such as drawing arcs between places and transitions, but you need text anyway to write the mathematical functions that govern firing of transitions. `YPetri` / `YNelson` are all textual without commiting the sin of introducing a new programming language. They are implemented as *internal domain-specific languages (DSL)* using Ruby as a host language. You can run them interactively from Ruby interpreter (*irb*). In this way, you can freely use the host language to automate your interaction with Petri nets.

`YPetri` / `YNelson` are publicly available as *y_petri* and *y_nelson gem* gems (*gem* = Ruby library). `YNelson` depends on `YPetri` and `Yzz` gems, which in turn depend on our `YSupport` gem. Splitting the more general concerns to multiple gem is a desirable feature – separation of concerns is a maxim of good software engineering. I also separated dealing with physical units in your models out into `SY` gem (*sy*).

Using This Guide, or "The Hard Way Is Easier"

The phrase above is borrowed from the textbook by Zed Shaw named "Learn Ruby the Hard Way" (highly recommended, *hyperlink here*). Apart from being a great shark-jumper, Zed is a great teacher familiar with many programming languages, and I will borrow his teaching method here. Citing Zed, "The title says it's the hard way... but actually it's not." It's only "hard" because of the way people *used* to teach things. `YNelson` is a language. To learn it and see its usefulness, you will still need to do the incredibly simple things that all language learners do:

1. Go through each example.
2. Type each sample code exactly.
3. Make it run.

That's it. This might feel *very* difficult at first, but stick with it. It seems stupidly obvious, but if you have a problem installing `YPetri` and `YNelson`, running *irb* and typing, you will have a problem learning. If you go through this document without actually doing the exercises, you might as well just not even read it. Do not skip and do not skim. By typing each example *exactly*, you will be training your brain to focus on the details of what you are doing, as you are doing it. While you do these examples, typing each one in, you will be making mistakes. It's inevitable; humans do make mistakes. By doing so, you will train yourself to notice mistakes and other problems. Do not copy-paste. Type each code sample in, manually. The point is to train your hands, your brain, and your mind in how to read, write and see Ruby and `YNelson` code. If you skip, skim and copy-paste, you are cheating yourself out of the effectiveness of this guide.

Prerequisites

Firstly, you will need a working installation of Ruby 2.3 or later on your computer. Once this condition is met, basic `YNelson` installation is as simple as typing "`gem install y_nelson`" in the command prompt. This will install most other dependencies, such as `YPetri`. However, `YNelson` currently uses dependencies (gnuplot gem, graphviz gem...), whose installation may pose challenges. Once `YNelson` is installed, run *irb* command interpreter, and type:

```
require 'y_nelson' and include YNelson
```

This will augment your irb command session with interactive `YNelson` command interface (`YNelson` DSL CI). You may also choose to work only with `YPetri` alone, in which case the require line is:

```
require 'y_petri' and include YPetri
```

You have to re-run *irb* from the scratch, and re-type '`require`' and '`include`' statements before each of the usage examples written below. The nucleotide metabolism model in Example 3 is not realiscit (its kinetics is not precise). If something in this guide does not work, please do not hesitate to notify me, I will appreciate your feedback.

Example I: Basics

This example is a gentle introduction to Petri net terminology, `YNelson` DSL terminology, and Ruby syntax. The most basic capability that `YNelson` offers, is that of user-driven *token game*. We will thus create a small Petri net containing 2 *places* and play token game with it.

Places

Type:

```
A = Place( marking: 2 )
```

Syntactically, this will call *method* '`Place`' of `YNelson` DSL and assign its *return value* to the *constant* `A`. In this case, the method is called with a single *argument*, `marking`. The return value is an *object*, which is an *instance* of `YNelson::Place` *class*. We say that `YNelson::Place` class *represents* the concept of Petri net places in `YNelson`

domain model. ('`Place`' method is called a *constructor* of `YNelson::Place`, which is not important.) This newly created Petri net place has now been assigned to `A`. On the screen, you will see the output, which is simply the name of the place – `A`. In the following text, screen output will always be written immediately under the code sample, preceded by sherocket (`#=>`):

```
B = Place()
#=> B
```

In the above, we have so far defined 2 Petri net places (named `A`, `B`). You can check them by typing:

```
places()
#=> [A, B]
A.name()
#=> :A
```

These have automatically become part of a default Petri net instance (of `YNelson::Net` class; object id may vary):

```
net()
#=> #<Net: name: Top, 2 places, 0 transitions>
```

Of course, you have full power of Ruby at your disposal. To eg. list only place names as strings, you can use standard Ruby methods:

```
places.map( &:name )
#=> [:A, :B]
```

Here, Ruby `map` method transforms the *array* of places to the array of their names. The advantage of internal DSLs is, that one retains full power of the language, augmented with human-friendly, domain-specific commands. GUI systems generally sandbox the user inside their interface, with no way to overcome its limitations. But let us go on. You can notice that the *marking* of `A`, `B` is one we gave them upon their creation:

```
places.map( &:marking )
#=> [2, 5]
```

In classical Petri nets, marking is understood as the number of *tokens* in each place, which is always integer. In this case, `A` contains 2 tokens, while `B` contains 5 tokens. Tokens can represent anything: molecules, parts in the production line, trains in the railway network...

Transitions

The behavior of a Petri net is defined by *transitions*. Each transition defines a single operation: Adding / subtracting some amount of tokens to / from some places. Transition operation can often be expressed by *stoichiometry* – pairs of places with a corresponding number of tokens to add / subtract when the transition *fires*. For example, let us define:

```
A2B = Transition( stoichiometry: { A: -1, B: 1 } )
#=> A2B
```

Stoichiometry of this transition is given by the *hash* `{ A: -1, B: 1 }`. This hash is available from `A2B` via '`s`' method:

```
A2B.s()
#=> {:A=>-1, :B=>1}
```

Keys of this hash are place names, *values* are *stoichiometry coefficients*. ('Stoichiometry' is a word known from the domain of chemistry, but '*stoicheion*' means simply 'element' in Greek, so there is no problem with using it in the domain of general Petri nets.) To see the stoichiometry coefficients of `A2B` as an array, type:

```
A2B.stoichiometry()
#=> [-1, 1]
```

Simply, `A2B` subtracts 1 token from `A`, and adds 1 token to `B`. This can represent conversion of `A` to `B`. In classical Petri nets, the arrows connecting places and transitions are called *arcs*. (The term was borrowed from graph theory.) For example, at this moment, our Petri net would contain one arc going from `A` to `A2B`, and one arc going from `A2B` to `B`. In `YNelson` domain model, 'arcs' are not first-class citizens. The word is understood simply as a synonym for transitions' connectivity – the list of places connected to each transition:

```
A2B.arcs()
#=> [A, B]
```

The transition `A2B` is *timeless*:

```
A2B.timeless?()
#=> true
```

'Timeless' means that the transition's firing is not defined in time – it can fire anytime, as long as it is *enabled*. Classical Petri nets are timeless. In classical Petri nets, a transition is enabled whenever its *downstream arcs* allow it to happen. Downstream arcs, or *codomain* of a transition (these two are synonyms) are those places, whose marking can be directly affected by the transition's firing. In this case, both `A` and `B` is affected:

```
A2B.downstream_arcs()
[A, B]
A2B.codomain()
[A, B]
```

Since `A2B` subtracts tokens from `A`, it will be enabled so long, as there are any tokens left in `A`.

```
A2B.enabled?()
#=> true
```

Token game

After `A2B` fires, the marking will change:

```
A2B.fire!()
#=> nil
places.map( &:marking )
#=> [1, 6]
A2B.fire!()
#=> nil
places.map( &:marking )
#=> [0, 7]
```

At this point, there are no tokens left in `A` and `A2B` becomes *disabled*:

```
A2B.enabled?
#=> false
```

Attempt to fire a disabled transition *raises* an *error* (in Ruby, errors are friendly objects, who, like damsels in distress, are meant to be rescued with a bonus outcome):

```
A2B.fire!
#=> YPetri::GuardError: When trying call #fire method, adding action node no. 0 to A, marking change
```

Functional transitions and non-integer marking

So far, all the examples were compatible with classical Petri nets. But `YNelson` goes beyond – it represents *functional Petri nets*, similar to *Hybrid Functional Petri Net (HFPN)* proposed by Matsuno et al. [2011], which I already mentioned in the introductory part of this document. `YNelson` domain model is similar, but not identical to HFPN. On the side of similarities, `YNelson` allows non-integer marking of places:

```
C = Place( marking: 7.77 )
#=> C
```

Here, you can notice that marking of places can be specified already upon initialization using '`:marking`' *named argument*. Let us now define a *timed* transition, representing logarithmic decay of `C` with a rate constant of 0.05:

```
C_decay = Transition( stoichiometry: { C: -1 }, rate: 0.05 )
#=> C_decay
C_decay.timed?
#=> true
```

Here, in the `transition` constructor method, apart from '`stoichiometry:`' named argument, another named argument, '`rate:`', is introduced. Under '`rate:`', it is possible to specify the transition's *function*, which governs its rate. Specifying a function in Ruby requires special syntax (called Ruby *closures*), based on lambda calculus. Ruby closures are easy to learn. But for the moment, in `C_decay` transition, we are taking use of the convenience, that allows us to pass a numeric value under '`rate:`' named argument, and have `YNelson` create default mass action equation, using the supplied number as its rate constant. For `C_decay` stoichiometry, `{ C: -1 }`, default mass action will be logarithmic decay with rate constant 0.05. Naturally, when firing timed transitions, the time interval (Δtime) must be specified, for which the transition should be active:

```
C_decay.fire!( 1 )
#=> nil
C.marking
#=> 7.3815
C_decay.fire! 1
#=> nil
C.marking
#=> 7.012425
C_decay.fire!( 0.1 )
#=> nil
C.marking
#=> 6.977362875000001
100.times do C_decay.fire! 1 end
#=> 100
C.marking
#=> 0.04130968078231133
```

The penultimate statement was a call of Ruby '`times`' method with the integer `100` as the receiver, which results in 100 time repetition of the statement inside `do ... end` *block*. Instead of `do ... end`, it is possible to write a block using curly braces `{ ... }`:

```
100.times { C_decay.fire! 1 }
#=> 100
```

This will cause another 100 time units of `C_decay` firing. This brings `C` marking down to almost zero:

```
C.marking
#=> 0.00024457517215434527
```

Four transition types

Thus far, we have demonstrated transitions with stoichiometry, which were either *timed* or not timed (*timeless*). Timed transitions are denoted by capital "`T`", timeless transitions by small "`t`". Similarly, stoichiometric transitions are denoted by capital "`S`", while transitions without stoichiometry (*non-stoichiometric* transitions) by small "`s`". Together, this gives 4 basic types of transitions: `TS`, `tS`, `Ts`, and `ts`.

The user can ask the type of a transition by calling the `type` method:

```
A2B.type
#=> :tS
```

Or investigate the type with inquirer methods:

```
A2B.t?
#=> true
A2B.T?
#=> false
A2B.s?
#=> false
A2B.S?
#=> true
A2B.TS?
#=> false
A2B.tS?
#=> true
A2B.Ts?
#=> false
A2B.ts?
#=> false
```

Assignment transitions

In `YNelson`, there is one more transition type: an assignment transition, denoted by "`A`". Assignment transitions do not add or subtract tokens from their target, but completely replace the codomain marking with their output. (Again, in `YNelson` transitions, *domain* and *codomain* mean respectively upstream and downstream places.) Transitions other than `A` transitions can be collectively called non-assignment transitions, denoted by small "`a`". Note that assignment action is already achievable with plain `ts` transitions (by subtracting away the previous codomain marking), so `A` transitions are not strictly needed – their separate existence is just a syntactic convenience.

One way to construct assignment transitions is by setting `:assignment` named argument to *true*:

```
A_to_42 = Transition codomain: A, assignment: lambda { 42 }
#=> A_to_42
```

Firing this transition results in marking of `A` being set to 42:

```
A_to_42.fire!
#=> nil
A.marking
#=> 42
```

Assignment transitions are of special type `A`:

```
A_to_42.type
#=> :A
A_to_42.A?
#=> true
A_to_42.a?
#=> false
```

Example II: Convenience

So far, to avoid confusing you, I used fairly conservative syntax. `YNelson` can do better than that. For convenience, many long keywords have short aliases. Frequently used syntactic constructs usually have shorter way of expressing the same.

As you have seen, in `YNelson` one often uses transition constructors. Thus far, we have seen only one *constructor method* for transitions: `Transition()`. `Transition()` method accepts several different named arguments (`:domain`, `:codomain`, `:stoichiometry` `:assignment`, `:rate`, `:action`, `:name`...) and depending on their values, returns a `YNelson::Transition` class object of required type and properties.

Using long keywords in the constructor method makes the `YNelson` code easy to read. But for the cases where trading readability for brevity is desirable, such as when you are playing with YNelson inside *irb* session, you will appreciate convenience. Actually, we already used syntactic shorthands in the earlier examples. We didn't type :

```
Transition( name: ‘‘A2B’’,
            codomain: [A, B],
            stoichiometry: [-1, 1] )
A2B = transition( :A2B )
```

Instead, we just typed

```
A2B = Transition( stoichiometry: { A: -1, B: 1 } )
```

Just so you know, the above shorthand is not easy to program in Ruby. When designing `YNelson`, I took a lot of pain for your convenience. Even shorter way to express the same would be:

```
A2B = Transition s: { A: -1, B: 1 }
```

The above is a very simple timeless transition that just takes one token from `A` and puts it into `B`. (Constructor convenience is even more powerful for complex transitions.) Start a new `irb` session and type:

```
require 'y_nelson'
include YNelson
A = Place default_marking: 5
```

Constantly typing `default_marking` is tiresome. Shorter way to say the same is by using the alias `m!` of the same:

```
B = Place m!: 5
C = Place m!: 1
D = Place m!: 1
```

Let's check the net's marking vector now:

```
places.map &:marking      #=> [5, 5, 1, 1]
```

If you typed everything correctly, you should see the above result. Shorter way to ask for the same information is:

```
net.marking                #=> [5, 5, 1, 1]
```

Even shorter way to say the same is:

```
net.m                      #=> [5, 5, 1, 1]
```

Now let's define the transition we want:

```
B2A = Transition( stoichiometry: { B: -1, A: 1 },
                  domain: [C, D],
                  rate: lambda { |x, y| ( x * y ) ** 0.5 } )
```

To prove that it works, let's fire it for 0.1 time units:

```
B2A.fire! 0.1              #=> nil
net.m                      #=> [5.1, 4.9, 1, 1]
```

Can we express its constructor more concisely? The answer is yes. `B2A` is a `TS` transition (check `B2A.type` to make sure it is true), and for `TS` transitions, `TS()` convenience constructor is available. With `TS()` constructor, the definition of B2A would be much shorter:

```
B2A = TS domain: [C, D], A: 1, B: -1 do |x, y| ( x * y ) ** 0.5 end
```

Note the `do ... end` part in the above line: Using lambda syntax, it defines the rate function of the transition. Another convenience constructor worth mentioning is `AT()` for assignment transition. Earlier, we defined:

```
A_to_42 = Transition codomain: A, assignment: lambda { 42 }
```

Using `AT()` constructor, we can shorten this to:

```
A_to_42 = AT A do 42 end
```

In short, syntactic shorthands can save a lot of typing. If you still miss some syntactic shorthand, feel free to define it on your own. For example, let us define a custom method named `Foo` that acts as a constructor of places with default marking.

```
def Foo( m )
  Place( default_marking: m )
end
```

After this, you can define places with default marking with even less typing:

```
X = Foo 42
Y = Foo 43
```

Example III: YNelson::Simulation

So far, we have been defining Petri nets and playing the token game using `#fire!` method, let us now simulate a Petri net inside `YNelson::Simulation`. Restart your irb session as described in the **Prerequisites** chapter. We will now define 2 places. Since we are going to use `TimedSimulation`, the marking owned by `YNelson::Place` instances is irrelevant. We just need to specify the initial state eg. by specifying the default marking of the places:

```
require 'y_nelson'
include YNelson
A = Place m!: 0.5
B = Place m!: 0.5
```

Now let us define a transition corresponding to pumping `A` out of the system at a constant rate 0.005 per time unit.

```
A_pump = Transition s: { A: -1 }, rate: proc { 0.005 }
```

Here, `proc { 0.005 }` is a closure, that defines the rate function. Closure `proc { 0.005 }` ensures fixed rate 0.005 per time unit regardless of the marking of `A`. You can notice that this closure expects no arguments and always outputs 0.005 as its return value. It is the simplest possible way to write a constant function. For comparison,

```
B_decay = Transition s: { B: -1 }, rate: 0.05
```

will behind the scenes automatically create a slightly more complicated mass action closure, which is logarithmic decay of `B` in this case. (You should remember this from **Example I**.) Now we have created a net of 2 places and 2 transitions:

```
net
#=> #<Net: name: Top, 2 places, 2 transitions>
```

We can execute this Petri net as a `Simulation` object simply by typing:

```
run!                #=> 60
```

At this point, `run!` creates and executes a `Simulation` instance. The return value is the simulation instance itself (see the inspect string above), which by now has already finished execution and holds the simulation results. This simulation instance is accessible via `simulation` method.

```
simulation
#=> #<Simulation: time: 60, pp: 2, tt: 2, oid: 75530290>
```

The simulation does not affect the net. The simulation instance works with its own "mental image" of the net, therefore the marking owned by `YNelson::Place` instances does not change:

```
net.marking         #=> [0.5, 0.5]
```

In a general case, it would be necessary to specify the simulation settings (step size, sampling rate, simulation time etc.) before running the simulation. Since we have not specified any, default settings were used:

```
simulation.settings
#=> {:method=>:basic, :guarded=>false, :step=>0.1, :sampling=>5, :time=>0..60}
```

We can see sampling done by the simulation by typing:

```
print_recording
#=> :A      :B
    ---------------
    0.5000  0.5000
    0.4750  0.3892
    0.4500  0.3029
    0.4250  0.2357
```

```
0.4000  0.1835
0.3750  0.1428
0.3500  0.1111
0.3250  0.0865
0.3000  0.0673
0.2750  0.0524
0.2500  0.0408
0.2250  0.0317
0.2000  0.0247
nil
```

Indeed, `A` is decreasing at a constant rate, while `B` undergoes logarithmic decay. In a graphical desktop, we can plot a graph (requires *gnuplot* gem):

```
recording.plot     # plots a graph
```

Previous command plots the default feature set, which is marking of the places. We can investigate also features of the recording (gradient or delta of places, firing or flux of the transitions...):

```
recording.gradient.plot
recording.flux.plot
recording.delta( delta_time: 0.1 ).plot
```

The last feature set – delta – requires `delta_time` named argument to extrapolate the changes (deltas) of the places in the given delta time. As for `firing`, a feature of `tS` transitions, the plot would show nothing here, as there ar no `tS` transitions here.

Example IV: A real system.

A highly simplified cell-biological pathway. Let's first define some assumptions. Type in the following commands (output not shown):

```
require 'y_nelson'
include YNelson
Pieces_per_microM = 100_000
set_step 10
set_sampling 30
set_target_time 30 * 60
```

Let's define places corresponding to chemical species first (note that `:m!` is a synonym for `:default_marking`)

```
AMP = Place m!: 8695.0
ADP = Place m!: 6521.0
ATP = Place m!: 3152.0
DeoxyCytidine = Place m!: 5.0
DeoxyCTP = Place m!: 20.0
DeoxyGMP = Place m!: 20.0
UMP_UDP_pool = Place m!: 2737.0
DeoxyUMP_DeoxyUDP_pool = Place m!: 10.0
DeoxyTMP = Place m!: 50.0
DeoxyTDP_DeoxyTTP_pool = Place m!: 100.0
Thymidine = Place m!: 10.0
```

All the places above have their marking in micromolars. The enzyme places below will have their marking in molecules per cell:

```
TK1 = Place m!: 100_000 / Pieces_per_microM
TYMS = Place m!: 100_000 / Pieces_per_microM
RNR = Place m!: 100_000 / Pieces_per_microM
TMPK = Place m!: 100_000 / Pieces_per_microM
```

Enzyme molecular weights:

```
TK1_kDa = 24.8
TYMS_kDa = 66.0
RNR_kDa = 140.0
TMPK_kDa = 50.0
```

Enzyme specific activities (in *micromolar* / *minute* / *mg*):

```
TK1_a = 5.40
TYMS_a = 3.80
RNR_a = 1.00
TMPK_a = 0.83
```

Some species are kept fixed (as simulation-level clamps):

```
clamp AMP: 8695.0, ADP: 6521.0, ATP: 3152.0
clamp DeoxyCytidine: 0.5, DeoxyCTP: 1.0, DeoxyGMP: 1.0
clamp Thymidine: 0.5
clamp UMP_UDP_pool: 2737.0
```

Before defining transitions, let's define some functions first:

```
Vmax_per_min_per_enz_molecule =
  lambda { |spec_act_microM_per_min_per_mg, kDa|
            spec_act_microM_per_min_per_mg * kDa }
Vmax_per_min =
  lambda { |spec_act, kDa, enz_molecules_per_cell|
           Vmax_per_min_per_enz_molecule.( spec_act, kDa ) *
             enz_molecules_per_cell }
Vmax_per_s =
  lambda { |spec_act, kDa, enz_mol_per_cell|
           Vmax_per_min.( spec_act, kDa, enz_mol_per_cell ) / 60 }
Km_reduced =
  lambda { |km, ki_hash={}|
           ki_hash.map { |c, ki| c / ki }.reduce( 1, :+ ) * km }
Occupancy =
  lambda { |c, km, compet_inh_w_Ki_hash={}|
           c / ( c + Km_reduced.( km, compet_inh_w_Ki_hash ) ) }
MM_with_inh_microM_per_second =
  lambda { |c, spec_act, kDa, enz_mol_per_cell, km, ki_hash={}|
            Vmax_per_s.( spec_act, kDa, enz_mol_per_cell ) *
              Occupancy.( c, km, ki_hash ) }
MMi = MM_with_inh_microM_per_second
```

Michaelis constants:

```
TK1_Thymidine_Km = 5.0
TYMS_DeoxyUMP_Km = 2.0
RNR_UDP_Km = 1.0
DNA_creation_speed = 3_000_000_000 / ( 12 * 3600 )
TMPK_DeoxyTMP_Km = 12.0
```

And finally, let us define the transitions:

```
Transition name: :TK1_Thymidine_DeoxyTMP,
           domain: [ Thymidine, TK1, DeoxyTDP_DeoxyTTP_pool, DeoxyCTP,
                     DeoxyCytidine, AMP, ADP, ATP ],
           stoichiometry: { Thymidine: -1, DeoxyTMP: 1 },
           rate: proc { |c, e, pool1, ci2, ci3, master1, master2, master3|
                        ci1 = pool1 * master3 / ( master2 + master3 )
                        MMi.( c, TK1_a, TK1_kDa, e, TK1_Thymidine_Km,
                              ci1 => 13.5, ci2 => 0.8, ci3 => 40.0 ) }
Transition name: :TYMS_DeoxyUMP_DeoxyTMP,
           domain: [ DeoxyUMP_DeoxyUDP_pool, TYMS, AMP, ADP, ATP ],
           stoichiometry: { DeoxyUMP_DeoxyUDP_pool: -1, DeoxyTMP: 1 },
           rate: proc { |pool, e, mono, di, tri|
                        c = pool * di / ( mono + di )
                          MMi.( c, TYMS_a, TYMS_kDa, e, TYMS_DeoxyUMP_Km ) }
Transition name: :RNR_UDP_DeoxyUDP,
           domain: [ UMP_UDP_pool, RNR, DeoxyUMP_DeoxyUDP_pool, AMP, ADP, ATP ],
           stoichiometry: { UMP_UDP_pool: -1, DeoxyUMP_DeoxyUDP_pool: 1 },
           rate: proc { |pool, e, mono, di, tri|
                        c = pool * di / ( mono + di )
                        MMi.( c, RNR_a, RNR_kDa, e, RNR_UDP_Km ) }
Transition name: :DNA_polymerase_consumption_of_DeoxyTTP,
           stoichiometry: { DeoxyTDP_DeoxyTTP_pool: -1 },
           rate: proc { DNA_creation_speed / 4 }
Transition name: :TMPK_DeoxyTMP_DeoxyTDP,
           domain: [ DeoxyTMP, TMPK, DeoxyTDP_DeoxyTTP_pool, DeoxyGMP, AMP, ADP, ATP ],
           stoichiometry: { DeoxyTMP: -1, TMPK: 0, DeoxyTDP_DeoxyTTP_pool: 1 },
           rate: proc { |c, e, pool, ci4, mono, di, tri|
                        ci1 = di
                        ci2 = pool * di / ( di + tri )
                        ci3 = pool * tri / ( di + tri )
                        MMi.( c, TMPK_a, TMPK_kDa, e, TMPK_DeoxyTMP_Km,
                              ci1 => 250.0, ci2 => 30.0, ci3 => 750, ci4 => 117 ) }
Transition name: :PhosphataseI,
           stoichiometry: { DeoxyTMP: -1, Thymidine: 1 },
           rate: 0.04
Transition name: :PhosphataseII,
           stoichiometry: { DeoxyTDP_DeoxyTTP_pool: -1, DeoxyTMP: 1 },
           rate: 0.01
```

The created net can be visualized by:

```
net.visualize
```

The simulation should work.

```
run!
```

State recording can be plotted by:

```
recording.plot
```

Flux of the transitions can be plotted by:

```
recording.flux.plot
```

Please note that although this system qualitatively represents part of the deoxynucleotide metabolism network, its behavior is not realistic, because the available kinetic constants are not precise.

Example V: Using SY.

Here, we'll take a look at using `YNelson` with SY metrology library. If you are experienced with biochemical modeling, then you surely know how big pain the physical units are. Also, in **Example III**, you might have noticed how much attention has been spent on units (in the assumptions, variable names, constant names...) You could have noticed messy unit conversion formulas. The aim of `SY` is to take care of all this, to relieve the modeler from the task of unit conversion, to clean up the model code, and let the modeler concentrate on the real issue.

`SY` metrology library

`SY` is publicly available as a Ruby gem '`sy`'. After installing it (`gem install sy`), type:

```
require 'sy'
```

Afterwards, your `Numeric` objects (that is, numbers) should respond to methods representing physical units:

```
1.m                   #=> #<±Magnitude: 1.m>
1.s                   #=> #<±Magnitude: 1.s>
1.kg.m.s(-2)          #=> #<±Magnitude: 1.N>
1.cm + 1.mm           #=> #<±Magnitude: 0.011.m>
```

The core of the trick is that instead of naked numbers, numbers become magnitudes (`SY::Magnitude`) of specified physical quantities:

```
1.m.quantity          #=> #<Quantity:Length>
1.cm.min⁻¹.quantity   #=> #<Quantity:Speed>
```

(You can type `1.cm.min(-1)` if you find it difficult to type Unicode superscript characters "`⁻¹`".) Magnitudes can be converted back to numbers with `amount` (alias `to_f`) method:

```
1.km.amount           #=> 1000.0
1.cm.to_f             #=> 0.01
```

Collaboration between `SY` and `YNelson`

In a fresh `irb` session, enter:

```
require 'sy'
require 'y_nelson'
include YNelson
A = Place m!: 3.mM
B = Place m!: 4.mM
A2B = Transition s: { A: -1, B: 1 }, rate: 0.05.s⁻¹
B_decay = Transition s: { B: -1 }, rate: 0.002.s⁻¹
```

We hereby specified marking and rate in physical units. Presently, `YNelson` is not able to simulate such nets, but we can play token game with it:

```
net.m    #=> [#<±Magnitude: 0.003.M>, #<±Magnitude: 0.004.M>]
A2B.fire! 1.s
B_decay.fire! 1.s
net.m    #=> [#<±Magnitude: 0.00285.M>, #<±Magnitude: 0.00414.M>]
```

Let us fire the 2 defined transitions for 100 seconds:

```
100.times do
  A2B.fire! 1.s
  B_decay.fire! 1.s
end
net.m    #=> [#<±Magnitude: 1.69e-05.M>, #<±Magnitude: 0.0058.M>]
```

Finally, let us inspect the resulting marking of `A` and `B` expressed in micromolars:

```
A.marking.in :µM       #=> 16.873508277951963
B.marking.in :µM       #=> 5797.976678013365
```

Example VI: Other simulation methods

At this moment, the default simulation method `basic` method. This method is also called is implicit Euler, because when simulating timed nets, it implies first-order Euler method. For timed nets, YNelson provides two other methods: Runge-Kutta 4th order method and Gillespie stochastic method. Demonstrating Gillespie method:

```
require 'y_nelson'
include YNelson
A = Place m!: 10
B = Place m!: 10
AB = Place m!: 0
AB_association = TS A: -1, B: -1, AB: 1, rate: 0.1
AB_dissociation = TS AB: -1, A: 1, B: 1, rate: 0.1
A2B = TS A: -1, B: 1, rate: 0.05
B2A = TS A: 1, B: -1, rate: 0.07
set_step 1
set_target_time 50
set_sampling 1
set_simulation_method :gillespie
run!
print_recording
plot_state
```

The state recording should show a random walk of the system state over the period of 50 time units.

References

`W. Bos. Modeling biological systems using Petri nets. 2008.`

Ina Koch. Petri nets in systems biology. *Software & Systems Modeling*, 14(2): 703-710, 2015.

H. Matsuno, Y. Tanaka, H. Aoshima, A. Doi, M. Matsui, and S. Miyano. Biopathways_representation_and_simulation_on_hybrid_functional_petri_net. *Stud Health Technol Inform*, 162:77-91, 2011. URL http://www.ncbi.nlm.nih.gov/pubmed/21685565.