Introduction to YNelson and YPetri (hands-on tutorial)

December 4, 2014

Introduction

This document is a hands-on guide to YNelson, *Nelson nets* and, partially, Ruby language. It is not assumed that the reader is familiar with any of these, though familiarity with Ruby syntax would be an advantage. This document can at the same time serve as a guide to YPetri, which is one of the two main components of YNelson, and which caters solely to the concerns of Petri net-based modelling of dynamical systems.

If you have never heard about Nelson nets, do not wonder: it is a semi-novel concept based on *Petri nets* crossed with Ted Nelson's *ZZ structures*. If you follow this guide closely, you will receive a concise and efficient introduction to each of these three. (Remark: *only the Petri net aspect defined by YPetri is covered in this version of this guide. ZZ structure aspect will be covered in the future versions of this manual.)* Newly introduced Ruby keywords and terms are highlighted in red, Petri net terms in green, and YNelson keywords and terms in blue throughout this document.

YNelson is a domain model and a simulator of Nelson nets, a specific universal type of *functional Petri nets* living in a *ZZ space*. YNelson is designed for the purpose of modelling and simulation of dynamical systems, especially biochemical systems. Modelling dynamical systems (including biochemical systems) is one of the typical applications of Petri nets. For a review of the various flavors of Petri nets used in modelling biochemical systems, see eg. Bos [2008]. YNelson introduces its own universal Petri net type, which is similar, but not identical with *hybrid functional Petri nets (HFPN)* introduced by Matsuno et al. [2011]. This universal Petri net is defined in YPetri gem, on which YNelson is based, and it purports to provide a universal Petri net abstraction and a DSL to rule them all and in Ruby bind them.

YNelson (and YPetri) is implemented as an *internal domain-specific language (DSL)* in *Ruby programming language*, which you can use in scripts, or access interactively from inferior Ruby interpreter (*irb*). It is publicly available as *y_nelson gem* (*gem* = Ruby library). YNelson belongs to a series of Ruby gems (YPetri, YNelson, YChem, YCell, Yzz, metrology library SY...), whose design intent is to bring ergonomy into biochemical modelling. YNelson depends on YPetri and Yzz gems, its usage together with SY might be desirable when dealing with physical units. At the time of writing this text, YNelson was the only publicly available internal DSL for modelling dynamic systems in Ruby. DSLs designed for the purpose of modelling are known from other languages, but they are generally *external*, not internal DSLs. They are sometimes denoted as domain-specific modeling languages (DSML), a term which also applies to YNelson.

All in all, if you want to seriously work with *complicated* Petri Nets, no tool can avoid taking you through the process of learning the language of the tool's interface. This language can be visual only partially. GUI Petri net modelling tools can provide visual commands for simple tasks, such as creating places and transitions, or drawing arcs between them, but a textual language is needed anyway for more complex tasks, such as writing mathematical functions that govern firing of transitions, or managing the Petri net in an automated manner. YNelson is all textual, but it does not introduce its own language. It relies on powerful and intuitive syntax of a widely adopted general-purpose computer language – Ruby. YNelson is open source and leverages on the freedom and respect for the user inherent to its host language, whose role in making YNelson good can hardly be overstated. Possible bugs or missing features will never get you as a YNelson user in a dead end situation.

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 `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

Most importantly, you will need a working installation of Ruby 1.9 on your computer. Once this condition is met, basic `YNelson` installation is as simple as typing "`gem install y_nelson`" in the command prompt. 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. Also, please notice that this guide itself is alpha stage, so the actual `YNelson` version you will be using may somewhat differ from this guide. Also, the nucleotide metabolism model in Example 3 is yet to be tuned to be realistic. If something in this guide does not work, please do not hesitate to notify us, we 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, 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.

Use of whole words in the constructor method makes the `YNelson` DSL very explicit. But for the cases, where trading readability for brevity is desirable, these syntactic constructs can be shortened. Actually, we have already used this convenience 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 } )
```

Even shorter way to express the same would be:

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

The above is a timeless transition. But we could think eg. about a more complicated transition, that would transfer tokens from `B` to `A` with rate depending on the square root of the product of marking of `C` and `D`. Start a new `irb` session and type:

```
require 'y_nelson'
include YNelson
A = Place( default_marking: 5 )
B = Place m!: 5 # notice "m!" alias for "default marking"
C = Place m!: 1
D = Place m!: 1
```

Let's check our work:

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

Indeed, the net state has been set according to the default markings of the places. 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 } )
#=> B2A
```

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

```
B2A.fire! 0.1
#=> nil
places.map &:marking
#=> [5.1, 4.9, 1, 1]
```

You can try to change marking of C and D to control the rate:

```
[A, B].each &:reset_marking
C.marking = 4
D.marking = 9
places.map &:marking
#=> [5, 5, 4, 9]
B2A.fire! 0.1
places.map &:marking
#=> [5.6, 4.4, 4, 9]
```

We can see that the rate of `B2A` has risen 6 times as expected (4 * 9 is 36), so `B2A` works. The question is, could we have written `B2A` more concisely? For `TS` transitions (check `B2A.type` to make sure that it's a `TS` transition), `TS()` constructor is available, allowing to express the same transition with a shorter syntactic construct:

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

Restart the `irb` session again and use this shorter construct to see that the resulting transition behaves like before. Note the `do ... end` part of the construct: Using lambda syntax, it defines the rate function of the transition.

One more convenience constructor I want to mention here is `AT()` constructor for assignment transition. Earlier, we defined:

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

This can be conveniently rewritten using `AT()` constructor as:

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

In short, syntactic shorthands are less readable than full `Transition()` statements, but can save a lot of space and typing. In any case, in Ruby, the user can easily defined new aliases and routines that make the frequent tasks easier to type.

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. One way to do this is by specifying `:default_marking` named argument:

```
A = Place( default_marking: 0.5 )
#=> A
B = Place( default_marking: 0.5 )
#=> B
```

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( stoichiometry: { A: -1 }, rate: proc { 0.005 } )
#=> A_pump
```

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( stoichiometry: { B: -1 }, rate: 0.05 )
#=> B_decay
```

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 pp, 2 tt>
```

We can execute this Petri net as `TimedSimulation` simply by typing:

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

At this point, `run!` creates and executes a `TimedSimulation` 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:

```
places.map &: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=>:pseudo_euler, :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 simulated with `YNelson::TimedSimulation`. Let's first define some assumptions. Type in the following commands (output not shown):

```
require 'y_nelson' and 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
```

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 in the heel 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' and include YNelson
A = Place m!: 3.mM
#=> A
B = Place m!: 4.mM
#=> B
A2B = Transition s: { A: -1, B: 1 }, rate: 0.05.s⁻¹
#=> A2B
B_decay = Transition s: { B: -1 }, rate: 0.002.s⁻¹
#=> B_decay
```

Now we have created places and transitions, whose marking and rate closures are defined in physical units. Presently, `YNelson::TimedSimulation` will not accept such Petri net, so the only thing we can do is play the token game ourselves:

```
fire_both_transitions = proc { |delta_t|
  A2B.fire! delta_t
  B_decay.fire! delta_t
}
#=> #<Proc:0x9b48f1c@(irb):19>
```

Here, we have defined a closure accepting one argument Δt, which it will use to `fire!` both `A2B` and `B_decay`. By calling this closure repeatedly, we can simulate the network without use of `TimedSimulation`:

```
places.map &:marking
#=> [#<±Magnitude: 0.003.M>, #<±Magnitude: 0.004.M>]
fire_both_transitions.( 1.s )
#=> nil
places.map &:marking
#=> [#<±Magnitude: 0.00285.M>, #<±Magnitude: 0.00414.M>]
100.times do fire_both_transitions.( 1.s ) end
#=> 100
places.map &:marking
#=> [#<±Magnitude: 1.69e-05.M>, #<±Magnitude: 0.0058.M>]
A.marking.in :µM
#=> 16.873508277951963
B.marking.in :µM
#=> 5797.976678013365
```

Example VI: Other simulation methods

At this moment, the default simulation method is implicit Euler (or `pseudo_euler` – *pseudo* because timeless transitions and assignment transtitions also fire at each step in time). From other simulation methods, Gillespie

algorithm is available:

```
require 'y_nelson' and 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 the random walk of the system state over 50 time units.

References

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

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.