YNelson & YPetri – User Manual

October 21, 2013

What is YNelson

`YNelson` is a Ruby library (gem) providing the domain model and simulator of *Nelson nets*, and the domain-specific language (DSL) to handle them. A *Nelson net* is a cross between a *Petri net*, and a *ZZ structure* (as described by Ted Nelson, ...). In accordance with this, `YNelson` is a combination of two major dependencies: `YPetri` gem, providing the the domain model of Petri nets, and `Yzz` gem, providing the domain model of ZZ structures. At the moment being, the focus of the development (and this user manual) is the Petri net model – `YPetri`. Since `YNelson` provides the same interface to the Petri net model as `YPetri`, this text can also serve as a user manual to plain `YPetri`. As for the ZZ structures, they are a promising data structure acting as a non-SQL relational database, but their use by YNelson is still under development – ZZ structures are so basic, that it's not clear, how deep in the object model should their usage go.

`YNelson` requires Ruby 2.1, and can be installed from the command line by "`gem install y_nelson`". The main dependencies, `Yzz` and `YPetri`, should install automatically, but can be installed manually by "`gem install y_petri`" and "`gem install yzz`".

Functional Petri nets

Petri nets were described by C. A. Petri in the middle of 20th century. A Petri net can be used to represent various "wiring diagrams" – such as production lines, rail networks, electronic circuits, computer architectures, parallel execution diagrams, or chemical systems. In fact, already Petri himself proposet to use Petri nets for modeling chemical reactions. In other words, chemical network modeling was one of the design goals of Petri nets since their inception.

A Petri net diagram consists of *places*, *transitions*, and *arcs*. Places are typically drawn as circles, transitions as rectangles, and arcs as lines or arrows connecting places and transitions. Petri net places may contain *tokens*. Another word for a number of token in a place(s) is *marking*. In `YNelson`, the amount of tokens (place's marking) is represented by a number attached to the place. When transitions operate (*fire*), tokens are added to or removed from the places, to which the transition is connected. It is the property of Petri nets, that their *state* is fully expressed by the places' marking – the transitions are completely stateless.

Original Petri nets were *timeless*. In a timeless Petri net, transition firing is a discrete event, whose exact timing is not specified. Instead, a timeless Petri net specifies the conditions, under which a transition is *enabled* – *allowed* to fire, or *disabled* – is *prohibited* to fire. Typically, a transition in a timeless Petri net is generally enabled, when its input arcs have enough tokens for firing to happen, though other firing conditions may be specified. Timeless Petri nets are generally used to investigate concurrency – things like race conditions, network congestions, state reachability etc. Interaction with timeless Petri nets is called *token game*.

Timed Petri nets operate in time. They specify when (or how rapidly) the transitions firing occurs. (The specification may be of stochastic.) Whereas a timeless Petri net requires the user to play the token game, timed Petri net can be *executed* in time. Timed Petri nets actually represent a wiring diagram of a dynamic system, from which a set of differential equations describing the system can be derived. Even if the differential equations are not expressed explicitly, execution of a timed Petri net is tantamount to numeric integration of its equivalent DE system.

YPetri object model

`Place`, `Transition` and `Net` class

The main classes of `YPetri` object model are `Place`, `Transition`, and `Net`. A `Net` instance represents a Petri net, consisting of `Place` instances and `Transition` instances. `Net` class is a (subclass of) `Module` class, so places and transitions can simply be defined as its constants. If a transition belongs to a net, all the places connected to it must belong to the same net.

Transitions can be divided into ... s / S, t / T, a / A ... this gives 5 basic types ...

`Simulation` and `Core` class

`Simulation` class represents a simulation of a particular network, with particular simulation settings. `Core` class represents the simulator – a machine that performs the execution of a Petri net. When a simulation instance wants to proceed in time to a next state, it realies on a `Core` instance to perform the computation.

`Simulation` – Public instance methods

place access:

...

transition access:

...

element access:

...

inital marking access:

...

marking clamp access:

...

marking vector access:

m_vector – as a column vector

m – as an array

place_m – as place => value hash

p_m – as place name => value hash

pm – pretty prints the name => value hash

update_m – modifies the marking vector

marking_vector – as m_vector, for free places only

marking – as m, for free places only

place_marking – as place_m, for free places only

p_marking – as p_m, for free places only

update_marking – as update_m, but the places must be free places only

increment_marking – increments the marking vector with a supplied difference vector (whose positions corresponds to free places)

...

own methods:

firing – returns firing of the indicated tS transitions

t_firing – same as has transition name => firing

pfiring – pretty prints firing

??? flux

??? t_flux

??? pflux

??? gradient
??? p_gradient
??? pgradient
??? delta
??? p_delta
??? pdelta
...
Core class is a bit intimate to YPetri, it is not directly controlled by the user.
`Core` – Public class methods
new – constructor
using_simulation_method(symbol, guarded: false) – constructor
`Core` – Public instance methods
guarded?
delta_timeless
delta_tS
delta_ts
firing_vector_tS
increment_marking_vector(delta)
assignment_transitions_all_fire!

`Net::State`, state features, and `Record`

A particular Petri net, as mentioned earlier, may have state, which is entirely given by the marking vector of its places. This state is represented in YPetri by Net::State class. Furthermore, a net in a particular state can have various features defined on it. A feature is represented by `Net::State::Feature` class. At the moment being, following features are supported: `Marking`, `Firing`, `Flux`, `Gradient` and `Delta`. For each feature, measurement can be performed on a net in a particular state, returning the feature's value. A collection of such features is called a feature set, and is represented by `Net::State::Features` class in `YPetri`. Measurement performed for a particular feature set results in a `Record` instance. Record is an Array subclass, but it remembers the features to which the stored values correspond.

`Feature` – Public instance methods
extract_from(target, **nn) – extracts the feature from the given target object. The target object must own a net state for the same net, for which this state feature was defined.
label – label for this feature (to use in graphics etc.)
...
`Features` – Public class methods
new – Constructor / instance reference
marking, flux, firing, gradient, delta – Features instance reference
`Features` – Public instance methods
extract_from(target, **nn) - extracts the feature set from a given target, returning a Record instance.
new_record(values) – Record instance constructor
+, -, * – same as for arrays
labels – feature labels
reduce_features –
marking, flux, firing, gradient, delta – feature reduction
...
`State` – Public class methods
feature – Feature instance reference
features – Features (feature set) instance reference
marking, flux, firing, gradient, delta – Features instance reference
`State` – Public instance method
...

`Record` – Public class methods
features – returns the features to which the record values correspond
load(values) - constructs a new Record object from a given collection of values.
Record – Public instance methods
marking, firing, flux, gradient, delta – feature extraction from a record
fetch(feature) – single feature extraction from a record
dump – should it be renamed to #to_a?
print – pretty prints the records with feature names
state(marking_clamps: {}) – returns the state instance implied by the record and the supplied complementary set of marking clamps
reconstruct(**settings) – reconstructs a simulation instance from the record and the given simulation settings
euclidean_distance(other) – computes the Euclidean distance to another record with the same feature set.

`DataSet` class

...
`DataSet` – Public instance methods
net – a dataset knows its net
reduce_features – selects certain columns (features) of a dataset.
slice – since DataSet is a Hash subclass, YSupport's Hash#slice method is available to it, and can be used to select only certain rows (records) of a dataset.
marking, firing, flux, gradient, delta – additional convenience methods for selecting particular columns and rows from the dataset
record(event) – returns the record instance for a given recorded key
floor(event), ceiling(event) – given a key value, they return the nearest record actually present in the dataset – floor returns the nearest lower or equal, ceiling the nearest higher or equal record.
interpolate(event) – Interpolates a record for a given key.
print – pretty-prints the dataset
plot – plots the dataset
resample – resamples the recording
series – returns the data series for the specified features
timed? – whether the dataset type is timed
reconstruct(event:, **settings) – reconstructs a simulation instance for a given key

`YPetri::World` and `YPetri::Agent` class

`YPetri::World` is the object space where places, transitions, and nets live. (Originally, this class was named `Workspace`, but `World` is shorter.) As for `YPetri::Agent` class, it represents the user and acts as user proxy to `YPetri::World` instance. `YPetri::Agent` does not encapsulate `YPetri` as completely, as the Law of Demeter might require, but it does one very useful thing: It defines the top-level DSL commands (methods) that are available upon calling '`include YPetri`' or '`include YNelson`'.