Object model of YNelson and YPetri

June 28, 2016

`YNelson` and `YPetri`

`YNelson` is a Ruby library (gem) providing a domain model and a simulator of *Nelson nets*. A *Nelson net* is a cross between a *Petri net*, and a *ZZ structure*. For formal definition of the ZZ structure, see Dattolo and Luccio [2009]. `YNelson` has two major dependencies: `YPetri` gem that provides Petri nets and `Yzz` gem that provides ZZ structures. `YNelson` provides the same interface as `YPetri`, so this document is (except for `Yzz` features) applicable also to `YPetri`.

You should also take a look at the tutorial, "Introduction_to_YNelson_and_YPetri". However, the tutorial is a hands-on guide, while this document describes the abstraction of `YPetri` / `YNelson`. The described gems need Ruby 2.3 or newer and can be installed by typing `gem install y_nelson` in your command prompt. (The dependencies will install automatically.)

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" – production lines, railway networks, electronic circuits, computer architectures, parallel execution diagrams, or chemical systems. In fact, Petri himself has designed Petri nets with chemical systems in mind.

A Petri net consists of *places*, *transitions*, and *arcs*. Places are typically drawn as circles, transitions as rectangles. Places and transitions are connected by arcs, drawn as lines or arrows. Places may contain *tokens*. The amount of tokens in a place(s) is called *marking*. When a transition operates (*fires*), tokens are added to or removed from its connected places. *State* of a Petri net is fully expressed by the places' marking.

Classical Petri nets are *timeless*. In timeless Petri nets, firing of a transition is a discrete event, whose exact timing is not specified. Instead, conditions are specified, under which the transition is allowed or prohibited to fire (*enabled* or *disabled*). Typically, transitions are enabled when their input arcs have sufficient amount of tokens left. However, different firing conditions may be specified. Timeless Petri nets are used to study concurrency – race conditions, network congestions, state reachability etc. Interaction with timeless Petri nets is called *token game*.

In *timed Petri nets*, it is specified when (or how rapidly) firing of the transitions occurs. Timed nets are thus not interactive and can be autonomously *executed* in time. Timed Petri nets actually represent a wiring diagram of a dynamic system. Under certain conditions, a set of differential equations (DE) describing the system can be extracted from this wiring diagram. Execution of a such Petri net is equivalent to numeric integration of its DE system.`YPetri` object model

Brief hands-on demonstration of the interface can be found in the document *Introduction to YNelson*. The purpose of this chapter is to describe the Petri net object model of `YPetri` gem in more detail.

Aspects of `YPetri`

`YPetri` has two main mutually intertwined concerns:

1. To provide active object model of Petri nets.
2. To provide simulation for the dynamic systems expressed as Petri nets.

Correspondingly, `YPetri` has 2 aspects catering to its 2 concerns:

1. Petri net aspect.
2. Simulation aspect.

Major classes of the Petri net aspect are `Place`, `Transition` and `Net`. Places have their own marking attribute, transitions know their connectivity, their functions, and they can be triggered to fire and modify marking of their connected places. `Net` is basically a specialized collection of places and transitions.

Simulation aspect is catered for by `Simulation` class, representing a simulation run, and `Core` class representing a simulator – a machine that runs the calculations needed to perform the simulation steps.

Workspace (`World` class, where places, transitions and nets live), and manipulator (`Agent` class that represents and assists the user) are straddled across both aspects of `YPetri`.

`Place` class

`YPetri::Place` class represents Petri net places. It includes `NameMagic` and is normally used as a parametrized subclass (PS) depending on `YPetri::World`. The key attribute of a place is its marking (variable `@marking`). Interface related to marking is:

- `#m`, alias `#value` – getter of `@marking` instance variable.
- `#marking` – convenience method that acts as `@marking` getter without arguments, but can be used to define guards if block is supplied to it.
- `#marking=`, alias `#value=`, alias `#m=` – setter of `@marking`.
- `#add` and `#subtract` that change the value of `@marking`.
- `#default_marking`, `#default_marking=` – getter and setter of place's default marking (`@default_marking` variable).
- `#has_default_marking?` – informs whether the place has default marking defined (`@has_default_marking` variable).
- `#reset_marking` – sets `@marking` to the default value.

Another important group of methods are those related to the place's connectivity – arcs. They are defined in the `YPetri::Place::Arcs` mixin included by `YPetri::Place`. In Petri net diagrams, arcs are the lines that connect places and transitions. In YPetri, there is no real "arc" object. For places, `#arcs` method simply returns the connected transitions, and vice versa, for transitions, `#arcs` method returns connected places. Overview of the most important `Place` instance methods inherited from the `Arcs` mixin is here:

- `#upstream_arcs`, alias `#upstream_transitions` – getter of `@upstream_arcs`.
- `#downstream_arcs`, alias `#downstream_transitions` – getter of `@downstream_arcs`.
- `#arcs` – a union of upstream transitions and downstream transitions.

- #aa – names of the connected transitions.
- #precedents – precedents in the spreadsheet sense. Places whose marking directly influences firing of the upstream transitions of this place.
- #dependents – dependents in the spreadsheet sense. Places whose marking is changed by firing of the downstream transitions of this place.

For the remaining methods, see the class documentation. Place can also have guards, statements that validate the marking and limit it to only certain values. At the moment, guards are not fully handled by the Simulation class.

Transition class

YPetri::Transition class represents Petri net transitions. They are "functional" in the sense that they may have mathematical functions attached to them, that govern their firing according to the current marking of the Petri net places. (Term "functional" has also been used in other meanings in conjunction with Petri nets.) YPetri::Transition class utilizes NameMagic mixin. YPetri::Transition class is normally used as a parametrized subclass (PS) depending on YPetri::World. The most prominent attribute of a Transition instance is its function. There are 4 basic types of transitions in YPetri:

- ts – timeless nonstoichiometric
- tS – timeless stoichiometric
- Ts – timed nonstoichiometric
- TS – timed stoichiometric

They arise by combining 2 basic qualities:

- *timedness*
- *stoichiometricity*

You can find more information in the documentation of YPetri::Transition class.

Net class

YPetri::Net class represents functional Petri nets. It includes NameMagic and is normally used as a PS depending on YPetri::World. It is basically a specialized collection of Place instances and Transition instances. A transition may only be included in a net if all the places connected to it belong to the same net. Net instances own 2 parametrized subclasses:

- #State – getter of @State, a PS of YPetri::Net::State
- #Simulation – getter of @Simulation, a PS of YPetri::Simulation.

Important instance methods include:

- #include_place – adds a place to the net
- #include_transition – adds a transition to the net
- #exclude_place – removes a place from the net

- #exclude_transition – removes a transition from the net
- #include_net alias #merge! – includes another net in this net
- #exclude_net – removes the elements of another net from this net
- #<< – includes an element in the net
- #+ – returns a new net containing the union of the operands' elements
- #- – returns a new net containing the elements of the receiver minus the operand
- #functional? – inquirer whether the net is functional
- #timed? – inquirer whether the net is timed
- #simulation – constructor of a new simulation of this net

A Net instance has its own state, and can be asked about place marking, transiton flux etc. It is also capable of drawing a diagram with Graphviz, using #visualize method. For full listing of methods, see the class documentation.

World class

YPetri::World is the space where places, transitions and nets live. Originally, this class was named Workspace, but World is shorter. Owns PS of Place, Transition and Net, stored respectively in @Place, @Transition and @Net instance variables. Their instances can be constructed with:

- #Place – constructor of instances of Place PS.
- #Transition – constructor of instances of Transition PS.
- #Net constructor – constructor of instances of Net PS.

World assets are divided into two mixins: YPetri::World::PetriNetAspect and YPetri::World::SimulationAspect. Important instance methods of PetriNetAspect are:

- #place – Place PS instance finder.
- #transition – Transition PS instance finder.
- #net – Net PS instance finder.
- #places – returns all Place PS instances.
- #transitions – returns all Transition PS instances.
- #nets – returns all Net PS instances.

Important instance methods of SimulationAspect are:

- #new_simulation – constructor of simulations.
- #simulation – Simulation instance finder.
- #simulations – getter of @simulations, a hash of simulation instances and their settings.

`World` instance has also 3 instance variables useful for simulations, `@clamp_collections`, `@initial_marking_collections` and `@simulation_settings_collections`. Each simulation requires an initial marking collection, a clamp collection, and a hash of simulation settings. In a world, these collections / settings can be named and stored in the above mentioned instance variables for later use in simulations. See the class documentation for more details and the accessor methods of these instance variables.

`Simulation` class

While `YPetri` places have their own marking and the transitions make it possible to play the token game interactively, for many reasons, it is desirable to be able to execute Petri nets automatically. `YPetri::Simulation` class represents such simulations. `Simulation` instances do not operate directly on the Petri nets from which they were constructed. Instead, they form a representation ("mental image") of the places and transitions of the underlying net. `Simulation` instances do not change the state owned by the underlying net. Instead, they have their own marking vector, which they modify using a chosen simulation method in the way that simulates firing of the transitions. A simulation owns multiple parametrized subclasses:

- `#Place` – getter of `@Place`, a PS of `Simulation::PlaceRepresentation`
- `#Transition` – getter of `@Transition, a` PS of `Simulation::TransitionRepresentation`
- `#Places` – getter of `@Places`, a PS of `Simulation::Places`, representing a collection of place representations.
- `#Transitions` – getter of `@Transitions`, a PS of `Simulation::Transitions`, representing a collection of place representations.
- `#Elements` – getter of `@Places`, a PS of `Simulation::Elements`, representing a collection of element (either place or transition) representations.
- `#PlaceMapping` – getter of `@PlaceMapping`, a PS of `Simulation::PlaceMapping`, a specialized `Hash` that maps the simulation's places to a set of some values.
- `#InitialMarking` – getter of `@InitialMarking`, a PS of `Simulation::InitialMarking`, which in turn is a subclass of `PlaceMapping`.
- `#MarkingClamps` – getter of `@MarkingClamps`, a PS of `Simulation::MarkingClamps`, which in turn is a subclass of `PlaceMapping`.

Simulation can be of two types: Timed or timeless. These two types are defined in two mixins, `Simulation::Timed` and `Simulation::Timeless`, with which the simulation instance is conditionally extended during its initialization, depending on its type. Simulation has a number of specialized instance methods defined in several mixins located inside the `Simulation` namespace (`Places::Access`, `Transitions::Access`, `Elements::Access`, `InitialMarking::Access`, `MarkingClamps::Access`, `MarkingVector::Access`). You can find their complete listing in the class documentation. Some of the instance methods are:

- `#reset!` – resets the simulation.
- `#run!` – runs the simulation.
- `#run_upto` – runs the simulation to a given time (same can be achieved by "`run( upto: ... )`").
- `#step!` – steps the simulation.
- `#settings` – returns all the settings for this simulation.
- `#core` – simulator currently in use.

- `#recorder` – recorder instance currently in use.
- `#tS_SM` – stoichiometric matrix for tS transitions.
- `#TS_SM` – stoichiometric matrix for TS transitions.
- `#dup` – duplicate of the receiver simulation, with the possibility to change time and/or other simulation settings.

Defined in `Simulation::Places::Access`:

- `#include_place?` – inquirer whether the simulation includes a specified place.
- `#p` – net's place identified by the argument.
- `#Pp` – net's places, expects single array argument.
- `#pp` – net's places, arbitrary number of arguments. If called without arguments, returns all the net's places.
- `#Free_pp`, alias `#free_Pp` – net's free places, single array argument.
- `#free_pp` – net's free places, arbitrary number of arguments. If called without arguments, returns all of them.
- `#Clamped_pp`, alias `#clamped_Pp` – net's free places, single array argument.
- `#clamped_pp` – net's free places, arbitrary number of arguments. If called without arguments, returns all of them.

Defined in `Simulation::Transitions::Access`:

- `#include_transition?` – inquirer whether the simulation includes a specified place.
- `#t` – net's transition identified by the argument.
- `#Tt` – net's transitions, expects array argument.
- `#tt` – net's transitions, arbitrary number of arguments.
- `#ts_Tt`, `#tS_Tt`, `#Ts_Tt`, `#TS_Tt`, `#A_Tt` etc. – net's transitions of the specified type. As signified by capitalized "`Tt`", these methods expects a single array argument.
- `#ts_tt`, `#tS_tt`, `#Ts_tt`, `#TS_tt`, `#A_tt` etc. – versions of tnet's transitions of the specified type. As signified by capitalized "`Tt`", these methods expects a single array argument.

Defined in `Simulation::Elements::Access` (word "element" simply stands for etiher place, or transition):

- `#include?` – inquirer whether the simulation includes a specified place or transition.
- `#e` – net's element identified by the argument.
- `#Ee` – net's elements identified by a single array argument.
- `#ee` – net's elements, arbitrary number of arguments.

Defined in `Simulation::InitialMarking::Access`:

- `#Initial_markings`, alias `#initial_Markings` – initial markings of given *free* places, single array argument.
- `#initial_markings`, alias `#initial_marking` – initial markings, arbitrary number of arguments identifying free places.

- `#Im` – starting markings of an array of *any* (not just free) places, as they appear right after reset. Expects a single array argument.
- `#im` – starting markings of any places, arbitrary number of arguments.

Defined in `Simulation::MarkingClamps::Access`:

- `#Marking_clamps`, alias `#marking_Clamps` – marking clamp values of given *clamped* places, single array argument.
- `#marking_clamps` – initial markings, arbitrary number of arguments.
- `#marking_clamp` – expects a single clamped place, returns its marking clamp.

Defined in `Simulation::MarkingVector::Access`:

- `#M_vector` alias `#m_Vector` – marking of the selected places returned as a column vector. Expects a single array argument.
- `#state` – getter of the simulation's state vector (`@m_vector` instance variable).
- `#m_vector` – marking of the selected places returned as a column vector, any number of arguments.
- `#M`, `#m` – array-returning varieties of `#M_vector` and `#m`.
- `#p_M` (alias `#P_m`), `#p_m` – hash-returning varieties of `#M_vector` and `#m`.
- `#Pm`, `#pm` – pretty printing varieties of `#M_vector` and `#m`.

Defined in `Simulation::Timed` and `Simulation::Timeless` mixins:

- `#timed?` – inquirer whether the simulation is timed
- `#Recorder` – getter of `@Recorder`, a PS of `Petri::Simulation::Recorder`, an object that performs sampling and recording during simulation.
- `#Core` – getter of `@Core`, a PS of `YPetri::Core`, a machine abstraction class.

Defined in `Simulation::Timed` mixin:

- `#time` – current simulation time
- `#initial_time` – initial time of the simulation
- `#target_time` – target time of the simulation (used when `#run!` is called without arguments)
- `#time_range` – range-returning alternative for `#initial_time` and `#target_time`.
- `#step`, `#step=` – getter and setter of the simulation step size
- `#default_sampling` – sampling period for the simulation
- `#Fluxes` – takes a single array of TS transitions and returns their fluxes under current marking.
- `#fluxes`, alias `#flux` – like `#Fluxes`, but takes an arbitrary number of arguments. If no arguments are supplied, returns all of them.
- `#T_fluxes` (alias `#t_Fluxes`), `#t_fluxes` (alias `#t_flux`) – hash-returning varieties of `#Fluxes` and `#fluxes`.
- `#pflux` – pretty-printing variety of `#t_flux`.

The above list of methods is not exhaustive. For full list of methods and their documentation, see the documentation of the `Simulation` class.

`Simulation::Recorder` class

`YPetri::Simulation::Recorder` is a class used exclusively as a PS dependent on `YPetri::Simulation` instances. Its has tow key attributes:

1. `#features` – getter of `@features`, containing the feature set to be recorded. Its class is a PS of `Net::State::Features`. It can be specified explicitly upon initialization or via `#reset!` method. By default, markings of the free places are used.

2. `#recording` – getter of `@recording`, containing the recording itself. Its class is a PS of `Net::DataSet`.

In the course of simulation, the recorder performs sampling: Upon occurence of certain events, it records the feature set and stores it in the `@recording` object. For timed simulations, events are typically specified by time (`@next_time` variable). Timeless simulations are not handled in the current version of `YPetri`, but it can already be said that events will be specified as conditions defined on the marking vector (`@next_event` variable). Recorder's checking for whether the sampling condition is fulfilled is triggered by the `:alert!` message. `Recorder.new` constructor takes the following named arguments:

- `features:` the feature set to record.

- `recording:` option to plug a pre-constructed dataset to the recorder.

In timed simulations, `Recorder.new` also accepts:

- `sampling:` sampling period

- `next_time:` the next sampling time

Important instance methods of `Recorder` are:

- `#new_recording` – constructs a new recording dependent on `@features`.

- `#reset!` – resets `@recording` and optionally changes `@features`.

- `#alert!` – recorder expects this message whenever the system state changes.

- `#sample!` – private method that performs sampling

`Core` class

`YPetri::Core` class is the abstraction for the simulator machine. Originally, it was named `Simulator`, but `Core` is shorter. When a `Simulation` instance wants to proceed in time to a next state, it relies on a `Core` instance to perform the computation. `Core` was separated from `Simulation` for the purpose of facilitating future use of different machines to run the simulation. At the moment, plain Ruby is used to compute the simulation steps. `Core` instance is generally not directly controlled by the user. `Core` provides certain some basic interface, on which its mixins defining the different simulation methods rely:

- `#flux_vector` – flux vector for the nets with only TS transitions.

- `#flux_vector_TS` – for mixed nets, returns flux vector for only TS transitions.

- `#firing_vector_tS` – firing vector of tS transitions.

- `#delta_tS` – delta state caused by tS transitions.
- `#delta_ts` – delta state caused by ts transitions.
- `#gradient` – total system gradient for free places.
- `#gradient_Ts` – gradient contribution by Ts transitions.
- `#gradient_TS` – gradient contribution by TS transitions.

For the purpose of controlling the marking vector, `Core` provides 2 instance methods:

- `#assignment_transitions_all_fire!` – fires all A (assignment) transitions.
- `#increment_marking_vector` – expects a delta vector of the same size as the marking vector as its singe argument, and increments the marking vector by it.

Simulation method mixins take flux, gradient, delta etc., and based on them, compute the overall delta state, change the marking vector and fire assignment transitions as defined by the simulation method, alerting the recorder when the state changes. At the moment, simulation methods include:

- `:euler` – 1st order method for nets with only T (timed) transitions. Mixin: `Core::Timed::Euler`.
- `:pseudo_euler` – Euler method adaptation for nets with timeless transitions. Mixin: `Core::Timed::PseudoEuler`.
- `:gillespie` – For nets with only T transitions. Mixin: `Core::Timed::Gillespie`.
- `:runge_kutta` – 5th order method for nets with only T transitions. Mixin: `Core::Timed::RungeKutta`. (*Not working at the moment!*)

`State` class

The state of a Petri net is entirely given by marking of its places. `Simulation` instances maintain their own marking vector holding the net state, but the net instance also has it's own state class, a PS of `YPetri::Net::State` < `Array`. `State` class is thus commonly used as a PS dependent on a `Net` instance, whose array positions correspond to the net's places. This net is available as a public class method on the `State` PS:

- `#net` – returns the net on which this State PS depends.

Each such `State` PS in turn owns 2 dependent parametrized subclasses:

- `#Feature` – getter of `@Feature`, containing a PS of `State::Feature`. When called with arguments, acts as alias of `#feature`.
- `#Features` – getter of `@Features`, containing a PS of `State::Features`. When called with a single array argument, this message acts as a feature set constructor / validator.

Other public class methods include:

- `#feature` – `@Feature` instance identifier.
- `#features` – `@Features` instance constructor. (See the class documentation for its full description.)

Instance methods include:

- `#to_record` – given clamped places, it returns a `Record` instance containing the marking of the free places. If no set of clamped places is supplied, it is considered empty.

- `#marking` – returns the marking of a single given place in this `State` instance.
- `#markings` – expects an arbitrary number of places, returns a plain array of their markings. If no arguments are given, returns all of them.

`Feature` class

A feature defines a measurement which can be performed on a net in some state to extract the feature's value. In `YPetri`, `Net::State::Feature` class is used as a PS dependent on a `State` PS (not on a `State` instance):

- `#State` – the `State` PS by which this `Feature` PS was parametrized.

In the present implementation, this class serves as a mother class for more specialized feature classes: `Marking`, `Firing`, `Flux`, `Gradient`, `Delta` and `Assignment`. These are defined as `Feature` subclasses on the namespace of `Feature` itself, but at the same time, a PS of each of them is owned by the Feature PS:

- `#Marking`, `#Firing`, `#Flux`, `#Gradient`, `#Delta`, `#Assignment` – these public class methods defined on a PS of Feature returns the dependent parametrized subclasses for the classes of the same name (`Feature::Marking`, `Feature::Firing`, etc.)

Instance methods are defined inside these specialized feature subclasses, such as:

- `#extract_from` – extracts the receiver feature from the target object, returning the feature's value.
- `#type` – feature type
- `#label` – feature label (for use in graphics etc.)

`Features` class – feature set

A collection of features is called a feature set. Measurement performed for a particular feature set results in a record. In `YPetri`, `Net::State::Features` is a subclass of `Array`, representing an array of features. Originally, it was named `FeatureSet`, but `Features` is shorter. It is used as a PS dependent on a `State` PS (*not* on a `State` instance). It's owning `State` PS can be accessed via `#State` class method. Such `Features` PS in turn owns subclasses of `Net::State::Features::Record` and `Net::Dataset`, which can be accessed via `#Record` and `#Dataset` class methods. Other class methods include:

- `#Marking`, `#Firing`, `#Flux`, `#Assignment` – constructors of a set of marking features, accepting single array argument.
- `#marking`, `#firing`, `#flux`, `#assignment` – versions of the above constructors, that accept any number of arguments, and return full corresponding feature sets if no arguments are given.
- `#Gradient` – constructor of a set of gradient features, accepting an array and an optional :transitions named argument.
- `#gradient` – version of the above constructor accepting any number of ordered arguments, and returning full gradient feature set if no ordered arguments are given.
- `#Delta` – constructor of a set of delta features, accepting an array and an optional :transitions named argument.
- `#delta` – version of the above constructor accepting any number of ordered arguments, and returning full delta feature set if no ordered arguments are given.

- `#[]` – constructor that takes either an arbitrary number of ordered arguments, or a field of named arguments (`:marking`, `:firing`, `:gradient`, `:flux`, `:delta`, `:assignment`), identifying a (possibly) mixed set of features.

Furthermore, class method `#new` is tweaked to make the returned instances own a PS of `Record` and a PS of `DataSet` *doubly parametrized* by the instance. Therefore, also instance methods include:

- `#Record` – a PS of `Features.Record` PS parametrized by the `Features` instance.
- `#DataSet` – a PS of `Features.DataSet` PS parametrized by the `Features` instance.

Other instance methods are:

- `#load` – delegated to the `Record` PS owned by the instance.
- `#extract_from` – extracts a feature set from the target object, returning a record.
- `#Record` alias `#load` – constructs an instance from an array of values. The values must corresponds to the receiver feature set.
- `#+`, `#-`, `#*` – addition, subtraction, `Array`-like multiplication for feature sets.
- `#labels` – array of feature labels.
- `#reduce_features` – expects an argument identifying a set of features that is a subset of the receiver feature set, and returns that feature subset.
- `#Marking`, `#Firing`, `#Flux`, `#Assignment` – selectors of a subset of the receiver feature set, accepting single array argument.
- `#marking`, `#firing`, `#flux`, `#assignment` – versions of the above selectors, that accept any number of arguments, and return full corresponding subsets if no arguments given.
- `#Gradient`, `#Delta`, `#gradient`, `#delta` – selectors analogical to the above mentioned, but also accepting an optional named argument `:transitions` qualifying the features to select.

Furthermore, `:Record` message is overloaded in such way, that when sent with an argument, it acts as an alias of `#load` record constructor.

`Record` class

A record is basically an array, that remembers the features to which its values correspond. `Net::State::Features::Record` class is typically used as doubly parametrized PS, dependent firstly on a `Features` PS, and then on its particular instance. The owning feature set is accessible via `#features` class and instance method. Other class methods include:

- `#load` – constructs a record from a given collection of values.

Other instance methods include:

- `#dump` – converts the record to a plain array, with optional :precision named argument.
- `#print` – pretty prints the record with feature names.
- `#fetch` – returns a feature.
- `#state` – constructs a state, using the receiver record, and a set of complementary marking clamps supplied in the argument.

- `#reconstruct` – reconstructs a simulation from the receiver record.
- `#Marking`, `#Firing`, `#Flux`, `#Assignment` – selects the values of the specified feature subsets from the receiver record. Expects a single array-type argument.
- `#marking`, `#firing`, `#flux`, `#assignment` – the versions of the above methods accepting any number of feature-identifying arguments.
- `#Gradient`, `#Delta`, `#gradient`, `#delta` – selectors analogical to the above mentioned, but also accepting an optional named argument `:transitions` qualifying the features to select.
- `#euclidean_distance` – takes another record of the same feature set as an argument and computes the Euclidean distance to it.

`DataSet` class

`YPetri::Net::DataSet` represents a sequence of records sampled from the underlying net using certain feature set. It is a subclass of `Hash`, whose keys represent sampling events, and whose values are plain arrays, from which corresponding `Record` instances can be fully reconstructed (via `Record.load` method). It is typically used as doubly parametrized PS, dependent firstly on a `Features` PS, and then on its particular instance. The owning feature set is accessible via `#features` class and instance method. Other class methods include:

- `#events` – alias for `Hash#keys` – dataset keys represent sampling events.
- `#reduce_features` – selects certain columns (features) of a dataset.
- `#timed?` – inquirer whether the dataset is timed
- `#record` – reconstructs the `Record` instance corresponding to the given event.
- `#floor` – the nearest event smaller or equal to the argument.
- `#ceiling` – the nearest event greater or equal than the argument.
- `#records` – returns an array of `Record` instances revived from the receiver's values.
- `#reconstruct` – recreates the simulation at the given event.
- `#interpolate` – interpolates the recording at the given point (event). Return value is the Record class instance.
- `#resample` – resamples the recording.
- `#distance` – computes the distance to another dataset.
- `#series` – returns the data series for the specified features.
- `#reduce_features` – reduces the dataset into another dataset with a different set of features. Reduction to a subset of features is always possible. Reduction to a set of features that is not a subset of the receiver's set of features is only possible if the former can be inferred from the latter. Generally, this is the case if net state can be reconstructed from a receiver's record. From this net state, the desired new feature set is then extracted.
- `#Marking`, `#Firing`, `#Flux`, `#Assignment` – selects the values of the specified feature subsets from the receiver record. Expects a single array-type argument.

- #marking, #firing, #flux, #assignment – the versions of the above methods accepting any number of feature-identifying arguments.
- #Gradient, #Delta, #gradient, #delta – selectors analogical to the above mentioned, but also accepting an optional named argument :transitions qualifying the features to select.
- #to_csv – outputs the dataset in the CSV format.
- #plot – plots the dataset.

Agent class

YPetri::Agent / YNelson::Agent are dumb agents that represent and assist the user. Originally, this class was named Manipulator, but Agent is shorter. Agent does provide textual user interface, but it does not completely encapsulate YPetri (YNelson) interface. Rather, it defines a number of top-level DSL commands (methods) available upon calling 'include YPetri' / 'include YNelson'. Agent is not a part of the core object model of YPetri / YNelson, and it is hotter than other parts of YPetri / YNelson. For basic use, see the tutorial ("Introduction to YNelson"). For detailed description of Agent's assets, see the class documentation.

References

Antonina Dattolo and Flaminia L Luccio. A formal description of zz-structures. In *1st Workshop on New Forms of Xanalogical Storage and Function. CEUR*, volume 508, pages 7-11, 2009.