Package 'linesolver'

November 28, 2026

Type Package

Title Queueing Network Models and Solvers (LINE)

Version 3.0.10.0

Author Giuliano Casale [aut, cre],
Imperial College London, QORE Lab [cph]

Maintainer Giuliano Casale <g.casale@imperial.ac.uk>

Description Build and solve queueing network models: open, closed and mixed networks, layered queueing networks and models in random environments. Models are declared in R and analysed by the LINE engine through a C application binary interface. Mean value analysis, normalising constant, continuous time Markov chain, fluid, matrix analytic, bounding and simulation solvers are available, together with the portable 'line-model' JSON interchange format shared with the MATLAB, Java, Python and C++ editions of LINE.

License BSD_3_clause + file LICENSE

URL https://line-solver.sourceforge.net, https://github.com/imperial-qore/line-solver

BugReports https://github.com/imperial-qore/line-solver/issues

Encoding UTF-8

Depends R (>= 4.0)

Imports R6, jsonlite, utils, stats

LinkingTo BH

Suggests testthat (>= 3.0.0), knitr, rmarkdown

SystemRequirements The LINE engine, either as a prebuilt libline_capi named by LINE_CAPI_LIB or LINE_CAPI_HOME, or built from the bundled sources, which needs a C++17 compiler and CMake (>= 3.16). LAPACK is taken from R. The 'ldes', 'jmt' and 'lqns' solvers (the latter also through 'qnsolver' for its 'qns.*' methods) additionally run external programs; see vignette(``installation'').

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Contents

api_invoke . . . 3
api_list . . . 3
AvgTable . . . 4
Cache . . . 4
capi_available . . . 6
capi_run . . . 7
capi_status . . . 7
ClassSwitch . . . 8
ClosedClass . . . 9
Delay . . . 10
Distributions . . . 10
enums . . . 13
find_solver . . . 14
Fork . . . 14
JobClass . . . 15
Join . . . 17
LayeredNetwork . . . 18
LayeredNetworkGenerator . . . 20
LineNode . . . 23
line_version . . . 26
load_model . . . 26
Logger . . . 27
model_index . . . 27
Network . . . 28
NetworkGenerator . . . 32
NetworkSolver . . . 35
OpenClass . . . 43
Place . . . 43
Queue . . . 45
Router . . . 46
RoutingMatrix . . . 47
save_model . . . 48
sc . . . 49
SelfLoopingClass . . . 49
serial_routing . . . 50
SignalClass . . . 50
Sink . . . 51
Solvers . . . 52
solver_meta . . . 55
Source . . . 55
tget . . . 57
Transition . . . 57
validate_model . . . 62

Index **63**

api_invoke *Call one engine API function directly*

Description

Call one engine API function directly

Usage

```
api_invoke(name, args = list(), arith = "double")
```

Arguments

name	The function, e.g. "pfqn_ca".
args	A named list of arguments. A matrix is sent row by row, so Q[i, j] reaches the engine as row i, column j.
arith	One of "double", "long double", "float128", "rational".

Value

The function's result; a matrix comes back as a list of rows.

api_list *The registry of API functions the engine carries*

Description

Every function ported to the C++ engine, with how it is reached. Those with access == "api" are callable through api_invoke; "library" ones take an argument no JSON document can carry (a callable, a random generator) and reason says which; "model" ones operate on a whole model and are reached through a solver's analyses instead. "unwired" names a wiring gap and is expected to be empty.

Usage

```
api_list(access = "api")
```

Arguments

access	Which rows to keep: any of "api", "library", "model", "unwired", or NULL for all of them.

Value

A data.frame with columns function, domain, arithmetic (comma-separated modes), access, reason and reference, one row per function.

AvgTable *An average table returned by a solver*

Description

A `data.frame` carrying the solve's provenance as attributes: the resolved `method`, the `arith`, the iteration count, and any warning the engine raised.

Cache *Cache node*

Description

A cache of 'n_items' items held in lists of capacities 'item_level_cap'. A request of an input class is served as a hit or a miss and leaves as the class 'set_hit_class' / 'set_miss_class' names.

Super class

`LineNode` -> `Cache`

Methods

Public methods:

- `Cache$new()`
- `Cache$set_hit_class()`
- `Cache$set_miss_class()`
- `Cache$set_read()`
- `Cache$set_access_prob()`
- `Cache$set_access_graph()`
- `Cache$set_admission_prob()`
- `Cache$set_item_sizes()`
- `Cache$set_cost_caps()`
- `Cache$set_retrieval()`
- `Cache$clone()`

`Cache$new()`: Create a cache.

Usage:

```
Cache$new(model, name, n_items, item_level_cap, replacement_strategy = "RR")
```

Arguments:

`model` Owning 'Network'.

`name` Node name.

`n_items` Number of cacheable items.

`item_level_cap` Capacity of each list (a vector for several lists).

`replacement_strategy` 'ReplacementStrategy' value, e.g. '"LRU"'.

`Cache$set_hit_class()`: Class a hit of 'cls' leaves as.

Usage:

```
Cache$set_hit_class(cls, out)
```

Arguments:

`cls` Input class.

`out` Output class.

`Cache$set_miss_class()`: Class a miss of 'cls' leaves as.

Usage:

```
Cache$set_miss_class(cls, out)
```

Arguments:

`cls` Input class.

`out` Output class.

`Cache$set_read()`: Item popularity of the requests of 'cls'.

Usage:

```
Cache$set_read(cls, dist)
```

Arguments:

`cls` Job class or its name.

`dist` A discrete distribution over the items, e.g. 'Zipf(1.4, n)'.

`Cache$set_access_prob()`: Per-class item access costs between lists: for each class in model order, a list with one (h+1) x (h+1) matrix per item.

Usage:

```
Cache$set_access_prob(p)
```

Arguments:

`p` List over classes of lists over items of matrices.

`Cache$set_access_graph()`: Item access graph between lists, shared by every class: one (h+1) x (h+1) matrix per item, or a single matrix used for every item.

Usage:

```
Cache$set_access_graph(G)
```

Arguments:

`G` Matrix or list of matrices.

`Cache$set_admission_prob()`: Probability of admitting a missed item.

Usage:

```
Cache$set_admission_prob(p)
```

Arguments:

`p` Probability.

`Cache$set_item_sizes()`: Storage cost of each item.

Usage:

```
Cache$set_item_sizes(sizes)
```

Arguments:

`sizes` Positive integers, one per item.

`Cache$set_cost_caps()`: Cap on the total cost held by each list (one value: cache-wide).

Usage:

```
Cache$set_cost_caps(caps)
```

Arguments:

`caps` Positive integers.

`Cache$set_retrieval()`: Delayed-hit retrieval: misses of ‘cls‘ are fetched through the named queues before they are served.

Usage:

```
Cache$set_retrieval(cls, queues, items = NULL, capacity = NULL)
```

Arguments:

`cls` Job class or its name.

`queues` Nodes (or names) the retrieval visits.

`items` Optional named list mapping a 0-based item id (as a string) to the name of the class that retrieves that item, e.g. `list("0" = "Retr0", "1" = "Retr1")`. Those classes are NOT created here: declare them first, as the cache tutorial does.

`capacity` Optional capacity of the retrieval system.

`Cache$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Cache$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`capi_available` *Whether the compiled C ABI is present in this installation*

Description

The package is useful without it: building a model and reading or writing `line-model` JSON is pure R. Solving is not. Every entry point that needs the engine calls this first and refuses by name rather than failing inside `.Call` with no explanation.

Usage

```
capi_available()
```

Value

TRUE when the native library loaded and its ABI version matches the one this package was compiled against.

---

capi_run *Run one* line-cli *argument vector in the embedded engine*

Description

The whole CLI vocabulary is reachable this way, because the vocabulary is the argument vector. This is the only native entry point in the package.

Usage

```
capi_run(argv, model = NULL, collect_json = TRUE, error = TRUE)
```

Arguments

argv	Character vector of arguments, without a program name.
model	Optional model document as a single string, delivered to the engine as if piped to the binary's standard input. NULL means no standard input at all, which is not the same as "".
collect_json	Whether to gather the emitted JSON documents. Turn it off for a readable-output call such as --help-all, where the documents would be empty anyway.
error	Whether a non-zero status raises. FALSE returns the result so the caller can inspect the failure, which is what find_solver() wants.

Value

A list with status, documents (a list of parsed JSON values), text, diagnostics and error.

---

capi_status *Why the C ABI is unavailable*

Description

Why the C ABI is unavailable

Usage

```
capi_status()
```

Value

A one-line explanation, or "available".

ClassSwitch *Class switch node*

Description

Class switch node

Super class

LineNode -> ClassSwitch

Methods

Public methods:

- ClassSwitch$new()
- ClassSwitch$set_class_switch_matrix()
- ClassSwitch$clone()

ClassSwitch$new(): Create a class-switch node.

Usage:

```
ClassSwitch$new(model, name, C = NULL)
```

Arguments:

model Owning model.

name Node name.

C Optional class-switch matrix.

ClassSwitch$set_class_switch_matrix(): Set the class-switch matrix.

Accepts either a square numeric matrix over the model's classes in declaration order, or a list keyed by source class then target class. Stored and emitted name-keyed, which is the schema's primary form and carries no index base.

Usage:

```
ClassSwitch$set_class_switch_matrix(C)
```

Arguments:

C A square matrix or a name-keyed list.

ClassSwitch$clone(): The objects of this class are cloneable with this method.

Usage:

```
ClassSwitch$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

ClosedClass *A closed job class*

Description

A closed job class

Super class

JobClass -> ClosedClass

Methods

Public methods:

- ClosedClass$new()
- ClosedClass$clone()

ClosedClass$new(): Create a closed class.

Usage:

```
ClosedClass$new(model, name, njobs, refstat, prio = 0)
```

Arguments:

model Owning model.

name Class name.

njobs Population.

refstat Reference station: a node object or its name.

prio Priority; 0 is the default.

ClosedClass$clone(): The objects of this class are cloneable with this method.

Usage:

```
ClosedClass$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Delay *Delay (infinite-server) station*

Description

Delay (infinite-server) station

Super classes

`LineNode` -> `ServiceNode` -> `Delay`

Methods

Public methods:

- `Delay$new()`
- `Delay$clone()`

`Delay$new()`: Create a delay station.

Usage:

```
Delay$new(model, name)
```

Arguments:

`model` Owning model.

`name` Node name.

`Delay$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Delay$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Distributions *Distributions*

Description

Constructors for the service, arrival and think-time distributions the LINE engine accepts. Each returns an inert object carrying the wire representation; nothing is fitted or derived in R.

Usage

```
Exp(lambda)

Erlang(lambda, k)

HyperExp(p, lambda)

Coxian(mu, phi)

Det(value)

Uniform(a, b)

Gamma(alpha, beta)

Pareto(alpha, scale)

Weibull(alpha, beta)

Lognormal(mu, sigma)

Normal(mu, sigma)

Immediate()

Disabled()

Replayer(fileName, mean = NULL)

Geometric(p)

Bernoulli(p)

Binomial(n, p)

Poisson(lambda)

DiscreteUniform(min, max)

Zipf(s, n)

DiscreteSampler(p, x = NULL)

EmpiricalCDF(x, F)

NHPP(breakpoints, rates, cyclic = TRUE)

MMPP2(lambda0, lambda1, sigma0, sigma1)
```

```
ME(alpha, A)

RAP(H0, H1)

DMAP(D0, D1)

MAPt(breakpoints, D0, D1, cyclic = TRUE)

PHt(breakpoints, alpha, S, cyclic = TRUE)

MPH(alpha, S, exit)

MMAPt(breakpoints, D0, D1k, cyclic = TRUE)

MPHt(breakpoints, alpha, S, exit, cyclic = TRUE)

BMMAPt(breakpoints, D0, D1kb, cyclic = TRUE)

MAP(D0, D1)

PH(alpha, A)

APH(alpha, A)

MMAP(D0, D1k)

BMAP(D)

MarkedMMPP(D, K = length(D) - 1L)

Cox2(mu1, mu2, phi1)

Prior(distributions, probabilities)

PriorContinuous(param_dist, template, slots)
```

Arguments

`lambda`, `k`, `p`, `mu`, `phi`, `alpha`, `beta`, `sigma`, `a`, `b`, `scale`, `n`, `s`, `value`, `min`, `max`, `x`, `F`	Distribution parameters, as named by the 'line-model' schema.
`fileName`	Path to a trace file, for 'Replayer'.
`mean`	Optional mean, for 'Replayer'.
`breakpoints`, `rates`, `cyclic`	Piecewise-constant intensity schedule, for 'NHPP'.
`lambda0`, `lambda1`, `sigma0`, `sigma1`	Two-state modulated parameters, for 'MMPP2'.
`D0`, `D1`, `H0`, `H1`, `A`	Matrix parameters of the correlated point processes.

`S`, `exit`, `D1k`, `D1kb`
: Per-segment sub-generators, marked exit vectors, marked arrival blocks and marked BATCH blocks of the schedule families. Each is a list nested outermost-first: 'D1k[[mark]][[segment]]', and 'D1kb[[mark]][[batch]][[segment]]'.

`D`
: For 'BMAP', the block list 'D0, D1, ..., Dk' where 'Dj' releases a batch of 'j' jobs; for 'MarkedMMPP', 'D0' followed by one block per mark.

`K`
: Number of marks, for 'MarkedMMPP'.

`mu1`, `mu2`, `phi1`
: Phase rates and the completion probability after phase 1, for 'Cox2'.

`distributions`, `probabilities`
: The alternatives of a discrete 'Prior' and their weights.

`param_dist`
: The density of the uncertain parameter, for 'PriorContinuous'.

`template`
: A distribution built with 'params', whose parameter(s) named in 'slots' the engine replaces by each sampled value of the parameter.

`slots`
: Names of the parameters of 'template' the parameter fills.

Details

These are plain FUNCTIONS, so the idiom is 'Exp(1)' and not 'Exp$new(1)': the nodes and job classes are R6 generators and take '$new()', the distributions do not. Writing '$new' on one is refused by name rather than left to R's 'object of type 'closure' is not subsettable'.

'Gamma()' MASKS `stats::Gamma()`, the GLM family, and the package says so when it is attached. Both are reachable: write 'stats::Gamma()' for the family and 'Gamma(shape, scale)' for the distribution.

`enums` *Enumeration namespaces*

Description

Named lists whose members are the wire strings themselves, e.g. `SchedStrategy$FCFS` is `"FCFS"`. Unknown member names are refused rather than returning NULL.

Usage

```
SchedStrategy

RoutingStrategy

DropRule

ReplacementStrategy

ImpatienceType

TimingStrategy

DepartureDiscipline
```

find_solver *Which solvers can analyze a model, and why the others cannot*

Description

Which solvers can analyze a model, and why the others cannot

Usage

```
find_solver(model, metric = NULL, all = TRUE)
```

Arguments

model	A Network, or a model document as a string.
metric	Optional metric to require, e.g. "QLen".
all	When TRUE, include the combinations that cannot run, with the reason each was refused. That is usually the interesting half.

Value

A data.frame with solver, method, runnable, method_class, metrics and reason.

Fork *Fork node*

Description

Splits each arriving job into one task per outgoing link (times 'set_tasks_per_link'), to be recombined at the paired 'Join'.

Super class

LineNode -> Fork

Methods

Public methods:

- Fork$new()
- Fork$set_tasks_per_link()
- Fork$clone()

Fork$new(): Create a fork.

Usage:

```
Fork$new(model, name)
```

Arguments:

`model` Owning ‘Network‘.

`name` Node name.

`Fork$set_tasks_per_link()`: Tasks generated per outgoing link.

Usage:

```
Fork$set_tasks_per_link(n)
```

Arguments:

`n` Positive integer.

`Fork$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Fork$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`JobClass` *Base class for every job class*

Description

Not constructed directly; use ‘OpenClass‘, ‘ClosedClass‘ or ‘SelfLoopingClass‘.

Public fields

`model` The model this class belongs to.

`name` Class name, unique within the model.

`type` Schema class type: "Open", "Closed" or "SelfLooping".

`spec` Schema keys set on this class.

Methods

Public methods:

- `JobClass$new()`
- `JobClass$set_immediate_feedback()`
- `JobClass$set_deadline()`
- `JobClass$set_patience()`
- `JobClass$set_reference_class()`
- `JobClass$set_reply_signal_class()`
- `JobClass$set_wire_key()`
- `JobClass$print()`
- `JobClass$clone()`

`JobClass$new()`: Not called directly; use a concrete class type.

Usage:

```
JobClass$new(model, name, type, prio = 0)
```

Arguments:

`model` Owning model.

`name` Class name.

`type` Schema class type.

`prio` Priority; 0 is the default and the highest.

`JobClass$set_immediate_feedback()`: Keep a self-looping job of this class on the server it just used, at EVERY station. ‘sn.immfeed‘ is the OR of this class-wide spelling and the per-station ‘Queue$set_immediate_feedback‘.

Usage:

```
JobClass$set_immediate_feedback(on = TRUE)
```

Arguments:

`on` Whether to enable it; TRUE by default.

`JobClass$set_deadline()`: Due date of a job of this class, for EDD/EDF scheduling.

Usage:

```
JobClass$set_deadline(deadline)
```

Arguments:

`deadline` Relative deadline; Inf for none.

`JobClass$set_patience()`: Class-wide patience; a node-scoped patience overrides it.

Usage:

```
JobClass$set_patience(dist, impatience_type = "reneging")
```

Arguments:

`dist` Distribution, or NULL to remove.

`impatience_type` ‘"reneging"‘ (default) or ‘"balking"‘.

`JobClass$set_reference_class()`: Make this the reference class of its chain.

Usage:

```
JobClass$set_reference_class(on = TRUE)
```

Arguments:

`on` Whether it is.

`JobClass$set_reply_signal_class()`: Class that carries the reply of a reply signal.

Usage:

```
JobClass$set_reply_signal_class(cls)
```

Arguments:

`cls` Job class or its name.

`JobClass$set_wire_key()`: Set any ‘line-model‘ class key directly, in the schema’s form.

Usage:

```
JobClass$set_wire_key(key, value)
```

Arguments:

`key` Schema key of ‘JobClass‘.

`value` Value; NULL removes the key.

`JobClass$print()`: Print the class.

Usage:

```
JobClass$print(...)
```

Arguments:

`...` Ignored.

`JobClass$clone()`: The objects of this class are cloneable with this method.

Usage:

```
JobClass$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`Join` *Join node*

Description

Recombines the tasks of a job forked at ‘fork‘.

Super class

`LineNode` -> `Join`

Methods

Public methods:

- `Join$new()`
- `Join$set_strategy()`
- `Join$set_quorum()`
- `Join$clone()`

`Join$new()`: Create a join.

Usage:

```
Join$new(model, name, fork = NULL)
```

Arguments:

`model` Owning ‘Network‘.

`name` Node name.

`fork` The paired ‘Fork‘, or its name.

`Join$set_strategy()`: Join strategy.

Usage:

```
Join$set_strategy(strategy)
```

Arguments:

`strategy` ‘"STD"‘ (wait for every task) or ‘"PARTIAL"‘ (quorum).

`Join$set_quorum()`: Quorum of a partial join; sets the strategy to ‘"PARTIAL"‘.

Usage:

```
Join$set_quorum(k)
```

Arguments:

`k` Tasks needed to release the job.

`Join$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Join$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`LayeredNetwork` *A layered queueing network, as its ‘.lqnx‘ document*

Description

THIS IS A DOCUMENT AND NOT A MODEL BUILDER. The package builds flat `Network` models in R and reads `line-model` JSON back into them; it has no R-side LQN object graph, and inventing a half one that could only carry what the generator happens to produce would be worse than carrying the interchange the engine, MATLAB, the JAR and Python all already read. Write it out with `$save()` and hand the path to any of them.

Methods

Public methods:

- `LayeredNetwork$new()`
- `LayeredNetwork$lqnx()`
- `LayeredNetwork$name()`
- `LayeredNetwork$save()`
- `LayeredNetwork$print()`
- `LayeredNetwork$clone()`

`LayeredNetwork$new()`: Wrap an '.lqnx' document.

Usage:

```
LayeredNetwork$new(lqnx)
```

Arguments:

`lqnx` The XML, as a single string.

`LayeredNetwork$lqnx()`: The '.lqnx' document.

Usage:

```
LayeredNetwork$lqnx()
```

Returns: A single string.

`LayeredNetwork$name()`: The model name the document carries.

Usage:

```
LayeredNetwork$name()
```

Returns: A single string, or `NA` when the document names none.

`LayeredNetwork$save()`: Write the document to a file.

Usage:

```
LayeredNetwork$save(path)
```

Arguments:

`path` Destination path.

Returns: The path, invisibly.

`LayeredNetwork$print()`: Print a summary.

Usage:

```
LayeredNetwork$print(...)
```

Arguments:

`...` Unused.

`LayeredNetwork$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LayeredNetwork$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`LayeredNetworkGenerator` *A random layered queueing network generator*

Description

Draws an LQN of client reference tasks calling down through layers of server tasks hosted on processors. Each range is a two-element `c(lo, hi)` pair; an integer range rounds its bounds inwards, a real one is sampled uniformly between them. The defaults are all `c(1, 1)` with both infinity probabilities zero, which draws the deterministic skeleton.

Every draw goes through one seeded stream, so a seeded generator here reproduces the JAR's `jline.gen.LayeredNetworkGenerator` model for model at the same seed and settings.

Methods

Public methods:

- `LayeredNetworkGenerator$new()`
- `LayeredNetworkGenerator$set_seed()`
- `LayeredNetworkGenerator$set_model_name()`
- `LayeredNetworkGenerator$set_population_range()`
- `LayeredNetworkGenerator$set_think_time_range()`
- `LayeredNetworkGenerator$set_task_multi_range()`
- `LayeredNetworkGenerator$set_proc_multi_range()`
- `LayeredNetworkGenerator$set_host_demand_range()`
- `LayeredNetworkGenerator$set_synch_call_range()`
- `LayeredNetworkGenerator$set_task_inf_probability()`
- `LayeredNetworkGenerator$set_proc_inf_probability()`
- `LayeredNetworkGenerator$spec()`
- `LayeredNetworkGenerator$generate()`
- `LayeredNetworkGenerator$print()`
- `LayeredNetworkGenerator$clone()`

`LayeredNetworkGenerator$new()`**:** Create a generator.

Usage:

```
LayeredNetworkGenerator$new(seed = NULL)
```

Arguments:

`seed` Optional seed. Omit for a generator that is not reproducible.

`LayeredNetworkGenerator$set_seed()`**:** Seed the stream every draw comes from.

Usage:

```
LayeredNetworkGenerator$set_seed(seed)
```

Arguments:

seed A whole number.

`LayeredNetworkGenerator$set_model_name()`: Name the generated model.

Usage:

```
LayeredNetworkGenerator$set_model_name(name)
```

Arguments:

`name` A single string.

`LayeredNetworkGenerator$set_population_range()`: Reference-task populations. Strictly positive.

Usage:

```
LayeredNetworkGenerator$set_population_range(r)
```

Arguments:

`r` `c(lo, hi)`.

`LayeredNetworkGenerator$set_think_time_range()`: Client think times. Zero means an immediate think.

Usage:

```
LayeredNetworkGenerator$set_think_time_range(r)
```

Arguments:

`r` `c(lo, hi)`.

`LayeredNetworkGenerator$set_task_multi_range()`: Multiplicities of the finite-server tasks.

Usage:

```
LayeredNetworkGenerator$set_task_multi_range(r)
```

Arguments:

`r` `c(lo, hi)`.

`LayeredNetworkGenerator$set_proc_multi_range()`: Multiplicities of the finite-server processors.

Usage:

```
LayeredNetworkGenerator$set_proc_multi_range(r)
```

Arguments:

`r` `c(lo, hi)`.

`LayeredNetworkGenerator$set_host_demand_range()`: Activity host demands. Zero means an immediate activity.

Usage:

```
LayeredNetworkGenerator$set_host_demand_range(r)
```

Arguments:

`r` `c(lo, hi)`.

`LayeredNetworkGenerator$set_synch_call_range()`: Mean number of synchronous calls on a call arc.

Usage:

```
LayeredNetworkGenerator$set_synch_call_range(r)
```

Arguments:

`r` `c(lo, hi)`.

`LayeredNetworkGenerator$set_task_inf_probability()`: Probability that a task is infinite-server rather than FCFS.

Usage:

```
LayeredNetworkGenerator$set_task_inf_probability(p)
```

Arguments:

`p` A probability.

`LayeredNetworkGenerator$set_proc_inf_probability()`: Probability that a processor is infinite-server rather than PS.

Usage:

```
LayeredNetworkGenerator$set_proc_inf_probability(p)
```

Arguments:

`p` A probability.

`LayeredNetworkGenerator$spec()`: The specification as it will be sent to the engine.

Usage:

```
LayeredNetworkGenerator$spec()
```

Returns: A named list; keys left at their default are absent.

`LayeredNetworkGenerator$generate()`: Draw a model.

Usage:

```
LayeredNetworkGenerator$generate(
  clients = 1,
  levels = 1,
  tasks = 1,
  processors = 1
)
```

Arguments:

`clients` Number of client (reference) tasks.

`levels` Number of task layers. At most `tasks`.

`tasks` Number of server tasks, spread over the layers.

`processors` Number of processors. At most `tasks`.

Returns: A `LayeredNetwork`.

`LayeredNetworkGenerator$print()`: Print the specification.

Usage:

```
LayeredNetworkGenerator$print(...)
```

Arguments:

... Unused.

`LayeredNetworkGenerator$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LayeredNetworkGenerator$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
g <- LayeredNetworkGenerator$new(seed = 23000)
g$set_host_demand_range(c(0.1, 2))
lqn <- g$generate(clients = 2, levels = 2, tasks = 5, processors = 2)
lqn$save("model.lqnx")

## End(Not run)
```

`LineNode` *Base class for every node*

Description

Not constructed directly. Holds the model back-pointer, the name, the schema node type, and the 'spec' list of schema keys the user has set.

Public fields

`model` The model this node belongs to.

`name` Node name, unique within the model.

`type` Schema node type.

`spec` Schema keys set on this node.

Methods

Public methods:

- `LineNode$new()`
- `LineNode$.set()`
- `LineNode$.set_per_class()`
- `LineNode$set_routing_strategy()`
- `LineNode$set_routing_weight()`
- `LineNode$set_routing_sq_choices()`
- `LineNode$set_state_prior()`
- `LineNode$set_wire_key()`
- `LineNode$print()`

- LineNode$clone()

`LineNode$new()`: Not called directly; use a concrete node type.

Usage:

```
LineNode$new(model, name, type)
```

Arguments:

`model` Owning model.

`name` Node name.

`type` Schema node type.

`LineNode$.set()`: Record a schema key. NULL deletes it.

Usage:

```
LineNode$.set(key, value)
```

Arguments:

`key` Schema key.

`value` Value, or NULL to unset.

`LineNode$.set_per_class()`: Record a per-class schema key.

Usage:

```
LineNode$.set_per_class(key, cls, value)
```

Arguments:

`key` Schema key holding a class-keyed map.

`cls` Job class object or name.

`value` Value.

`LineNode$set_routing_strategy()`: Per-class routing strategy. NOTE the wire puts this at MODEL level, keyed by node name and then by class name; it is set here because that is where a user reaches for it, and the serialiser hoists it. Only strategies other than PROB and RAND are emitted, since the routing matrix already captures those.

Usage:

```
LineNode$set_routing_strategy(cls, strategy)
```

Arguments:

`cls` Job class.

`strategy` A RoutingStrategy member.

`LineNode$set_routing_weight()`: Routing weight of ‘cls‘ towards node ‘dest‘ under WR-ROBIN.

Usage:

```
LineNode$set_routing_weight(cls, dest, weight)
```

Arguments:

`cls` Job class or its name.

`dest` Destination node or its name.

`weight` Positive weight.

`LineNode$set_routing_sq_choices()`: Number of sampled destinations ‘d‘ of SQ(d) routing for ‘cls‘.

Usage:

```
LineNode$set_routing_sq_choices(cls, d)
```

Arguments:

`cls` Job class or its name.

`d` Positive integer.

`LineNode$set_state_prior()`: Explicit state space of this node and a prior over its rows.

Usage:

```
LineNode$set_state_prior(space, prior = NULL)
```

Arguments:

`space` Matrix, one state per row.

`prior` Probability of each row; defaults to all mass on the first.

`LineNode$set_wire_key()`: Set any ‘line-model‘ node key directly, in the form the schema gives it, for a key this class has no typed setter for. A distribution inside the value is converted; a length-1 vector is written as a scalar unless wrapped in ‘I()‘. NULL removes the key.

Usage:

```
LineNode$set_wire_key(key, value)
```

Arguments:

`key` Schema key of ‘Node‘.

`value` Value.

`LineNode$print()`: Print the node.

Usage:

```
LineNode$print(...)
```

Arguments:

`...` Ignored.

`LineNode$clone()`: The objects of this class are cloneable with this method.

Usage:

```
LineNode$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

line_version *The version and capabilities of the embedded engine*

Description

Reports the build's optional features, so a user can be told what this installation will refuse BEFORE they run something it refuses. A build without HiGHS answers some bounding problems with a capped dense tableau instead of the sparse LP; a build without LAPACK refuses the eigenvalue and Schur routines by name.

Usage

```
line_version()
```

Value

A list of class `line_version` with `package` and `available`; with an engine also `cli`, `abi`, `features` and `blas`, without one `why`. Its print method shows them one per line.

load_model *Read a 'line-model' document*

Description

Read a 'line-model' document

Usage

```
load_model(filename)
```

Arguments

`filename`	Path to a 'line-model' JSON document, or the JSON itself.

Value

A 'Network'.

Logger *Logger node*

Description

Passes jobs through unchanged and records their passage times.

Super class

`LineNode` -> `Logger`

Methods

Public methods:

- `Logger$new()`
- `Logger$clone()`

`Logger$new()`: Create a logger.

Usage:

```
Logger$new(model, name, file_name = NULL)
```

Arguments:

`model` Owning 'Network'.

`name` Node name.

`file_name` Optional trace file name.

`Logger$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Logger$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

model_index *What a model is made of, as the engine sees it*

Description

Returns the node and class names, the node types, the station list and the two index maps between nodes and stations.

Usage

```
model_index(model)
```

Arguments

`model`	A `Network`, or a model document as a string.

Details

WHY THIS IS NOT COMPUTED IN R. "Which node is a station" is a LINE semantic: it depends on the node type, and on the model kind, and it has changed. R deriving it would make a fifth place for that rule to live and a fifth place for it to drift. Asking the engine costs one call and cannot disagree with the engine.

Value

A list with `nodes`, `nodeTypes`, `classes`, `classTypes`, `stations`, `nodeToStation`, `stationToNode` and `refstat`. Indices are 1-based, converted from the document's own 0-based form at this boundary and nowhere else.

`Network` *A queueing network model*

Description

The model container. Nodes and job classes REGISTER THEMSELVES with the model when constructed, so ‘Source$new(model, "Source")‘ mutates ‘model‘ and the caller sees it without any reassignment. That is the idiom the MATLAB, Java, Python and C++ editions all use, and R6's reference semantics are what make it work here. ‘print(model)‘ shows what has accumulated.

The model is a DOCUMENT BUILDER. It records what you set and derives nothing: no visit ratios, no chains, no station partition, no moment fitting. All of that belongs to the engine, and a second place that computed it would be a second place it could disagree.

Public fields

`name` Model name. Node and class names must be unique within it.

`extras` Model-level ‘line-model‘ keys besides nodes, classes and routing: finite capacity regions, rewards, global dependence, and any key a loaded document carried.

Methods

Public methods:

- `Network$new()`
- `Network$nodes()`
- `Network$classes()`
- `Network$get_node()`
- `Network$get_class()`
- `Network$index_of()`
- `Network$link()`

- Network$add_region()
- Network$set_reward()
- Network$set_global_dependence()
- Network$init_routing_matrix()
- Network$routing()
- Network$.register_node()
- Network$.register_class()
- Network$copy()
- Network$print()

`Network$new()`: Create an empty model.

Usage:

```
Network$new(name)
```

Arguments:

`name` Model name.

`Network$nodes()`: Nodes, in the order that fixes their indices.

Usage:

```
Network$nodes()
```

`Network$classes()`: Job classes, in the order that fixes their indices.

Usage:

```
Network$classes()
```

`Network$get_node()`: Look up a node by name or by 1-based index.

Usage:

```
Network$get_node(x)
```

Arguments:

`x` Name or index.

`Network$get_class()`: Look up a job class by name or by 1-based index.

Usage:

```
Network$get_class(x)
```

Arguments:

`x` Name or index.

`Network$index_of()`: The 1-based index of a node, as R counts.

Usage:

```
Network$index_of(node)
```

Arguments:

`node` A node object or its name.

`Network$link()`: Set the routing.

Usage:

```
Network$link(P)
```

Arguments:

`P` A RoutingMatrix, or the result of 'serial_routing()'.

`Network$add_region()`: Add a finite capacity region over a group of stations.

Usage:

```
Network$add_region(
  name,
  stations,
  global_max_jobs = -1L,
  class_max_jobs = NULL,
  drop_rule = NULL,
  class_weight = NULL,
  global_max_memory = -1L,
  class_max_memory = NULL
)
```

Arguments:

`name` Region name, unique within the model.

`stations` Nodes (or names) the region spans.

`global_max_jobs` Cap on the total jobs in the region (-1: none).

`class_max_jobs` Named vector 'class -> cap'.

`drop_rule` Named vector 'class -> DropRule' (e.g. "drop", "waitingQueue").

`class_weight` Named vector 'class -> weight' (memory per job).

`global_max_memory` Cap on the total memory (-1: none).

`class_max_memory` Named vector 'class -> memory cap'.

`Network$set_reward()`: Declare a reward evaluated by the CTMC solver on the state space ('analysis = "reward"').

Usage:

```
Network$set_reward(name, type, node, cls = NULL)
```

Arguments:

`name` Reward name.

`type` '"QLen"', '"Util"' or '"Blocking"'.

`node` Node whose state the reward reads.

`cls` Optional class; NULL aggregates over the classes.

`Network$set_global_dependence()`: Global (Whittle) rate scaling phi(n) over the whole network state, in the schema's 'GlobalDependence' form (stations, classes, slots, cutoffs and the tabulated values).

Usage:

```
Network$set_global_dependence(spec)
```

Arguments:

spec Named list in schema form, or NULL to remove.

`Network$init_routing_matrix()`: A fresh routing matrix over this model's nodes and classes.

Usage:

```
Network$init_routing_matrix()
```

`Network$routing()`: The routing currently set, or NULL.

Usage:

```
Network$routing()
```

`Network$.register_node()`: Register a node. Called by node constructors; not user API.

Usage:

```
Network$.register_node(node)
```

Arguments:

node The node registering itself.

`Network$.register_class()`: Register a job class. Called by class constructors.

Usage:

```
Network$.register_class(cls)
```

Arguments:

cls The class registering itself.

`Network$copy()`: Refuse to copy in place. See 'cloneable = FALSE' below for why '$clone()' does not exist either.

Usage:

```
Network$copy()
```

`Network$print()`: Print a summary.

Usage:

```
Network$print(...)
```

Arguments:

... Ignored.

`NetworkGenerator` *A random queueing network generator*

Description

Draws a connected, routing-ergodic ‘Network‘ to a specification: how many queues and delays, how many open and closed classes, which scheduling and routing strategies, and which service distributions. Rejection-sampled, so every model it returns has each closed chain visiting its own reference station.

SEED IT TO REPRODUCE. An unseeded generator starts from the engine’s default stream and draws a different model each call; ‘$set_seed(n)‘ (or the ‘seed‘ argument) makes the whole sequence a function of ‘n‘.

Methods

Public methods:

- `NetworkGenerator$new()`
- `NetworkGenerator$set_seed()`
- `NetworkGenerator$set_model_name()`
- `NetworkGenerator$set_sched_strat()`
- `NetworkGenerator$set_routing_strat()`
- `NetworkGenerator$set_distribution()`
- `NetworkGenerator$set_cclass_job_load()`
- `NetworkGenerator$set_varying_service_rates()`
- `NetworkGenerator$set_multi_server_queues()`
- `NetworkGenerator$set_random_cs_nodes()`
- `NetworkGenerator$set_multi_chain_cs()`
- `NetworkGenerator$set_topology()`
- `NetworkGenerator$spec()`
- `NetworkGenerator$generate()`
- `NetworkGenerator$print()`
- `NetworkGenerator$clone()`

`NetworkGenerator$new()`**:** Create a generator.

Usage:

```
NetworkGenerator$new(seed = NULL)
```

Arguments:

`seed` Optional seed. Omit for a generator that is not reproducible.

`NetworkGenerator$set_seed()`**:** Seed the stream every draw comes from.

Usage:

```
NetworkGenerator$set_seed(seed)
```

Arguments:

`seed` A whole number.

`NetworkGenerator$set_model_name()`: Name the generated model.

Usage:

```
NetworkGenerator$set_model_name(name)
```

Arguments:

`name` A single string.

`NetworkGenerator$set_sched_strat()`: Scheduling discipline for the queues.

Usage:

```
NetworkGenerator$set_sched_strat(strat)
```

Arguments:

`strat` One of `"randomize"`, `"fcfs"`, `"lcfs"`, `"sjf"`, `"ljf"`, `"siro"`, `"ps"`, `"dps"`, `"gps"` or `"hol"`.

`NetworkGenerator$set_routing_strat()`: Routing strategy for the classes.

Usage:

```
NetworkGenerator$set_routing_strat(strat)
```

Arguments:

`strat` One of `"randomize"`, `"prob"` or `"rand"`.

`NetworkGenerator$set_distribution()`: Service distribution family.

Usage:

```
NetworkGenerator$set_distribution(d)
```

Arguments:

`d` One of `"randomize"`, `"exp"`, `"erlang"` or `"hyperexp"`.

`NetworkGenerator$set_cclass_job_load()`: Closed-class population band.

Usage:

```
NetworkGenerator$set_cclass_job_load(load)
```

Arguments:

`load` One of `"randomize"`, `"high"`, `"medium"` or `"low"`.

`NetworkGenerator$set_varying_service_rates()`: Whether service rates vary across stations and classes.

Usage:

```
NetworkGenerator$set_varying_service_rates(v)
```

Arguments:

`v` `TRUE` or `FALSE`.

`NetworkGenerator$set_multi_server_queues()`: Whether queues may have more than one server.

Usage:

```
NetworkGenerator$set_multi_server_queues(v)
```

Arguments:

`v` `TRUE` or `FALSE`.

`NetworkGenerator$set_random_cs_nodes()`: Whether class-switch nodes are inserted on random arcs.

Usage:

```
NetworkGenerator$set_random_cs_nodes(v)
```

Arguments:

`v` `TRUE` or `FALSE`.

`NetworkGenerator$set_multi_chain_cs()`: Whether a class switch may cross chains.

Usage:

```
NetworkGenerator$set_multi_chain_cs(v)
```

Arguments:

`v` `TRUE` or `FALSE`.

`NetworkGenerator$set_topology()`: Connection topology.

Usage:

```
NetworkGenerator$set_topology(kind)
```

Arguments:

`kind` `"rand"` (a random strongly connected graph) or `"cyclic"` (a single cycle).

`NetworkGenerator$spec()`: The specification as it will be sent to the engine.

Usage:

```
NetworkGenerator$spec()
```

Returns: A named list; keys left at their default are absent.

`NetworkGenerator$generate()`: Draw a model.

Usage:

```
NetworkGenerator$generate(
  queues = 1,
  delays = -1,
  open_classes = 0,
  closed_classes = 1
)
```

Arguments:

`queues` Number of queueing stations.

`delays` Number of infinite servers. `-1`, the default, lets the generator split the stations itself.

`open_classes` Number of open classes.

`closed_classes` Number of closed classes.

Returns: A Network.

`NetworkGenerator$print()`: Print the specification.

Usage:

```
NetworkGenerator$print(...)
```

Arguments:

`...` Unused.

`NetworkGenerator$clone()`: The objects of this class are cloneable with this method.

Usage:

```
NetworkGenerator$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## Not run:
g <- NetworkGenerator$new(seed = 23000)
g$set_varying_service_rates(TRUE)
m <- g$generate(queues = 3, open_classes = 1, closed_classes = 1)
print(m)

## End(Not run)
```

NetworkSolver *The shared surface of every solver*

Description

Not constructed directly. `MVA$new(model)` and its siblings are the entry points.

Methods

`avg_table()` The per-station, per-class average table.

`avg_sys_table()` The per-chain system table.

`avg_node_table()` The per-node table, which includes the non-station nodes a station table omits.

`avg_chain_table()` The per-chain station table.

`analysis(name, ...)` Any other `-a` analysis the CLI serves, by name. This is what makes the whole vocabulary reachable.

`get_avg()` The six metrics as (station x class) matrices.

`method_used()` The method the solve actually resolved to, which is not always the one asked for.

`list_valid_methods()` The methods this solver advertises on this model.

Typed getters

One method per analysis the engine serves, each a named spelling of `analysis(name, ...)` returning the engine's own payload (a parsed JSON list) unless `raw = FALSE` asks for a table. They take the same knobs as the solver (`tspan`, `node`, `class`, `state`, `events`, `busyperiod`, ...). The table is `SOLVER_GETTERS`; a solver that does not serve an analysis refuses it by name, which is the engine's decision and not a second copy of it here.

`get_avg_node_chain_table()` `-a nodechain`

`get_cdf_resp_t()` `-a cdf`

`get_cdf_pass_t()` `-a cdf-passt`

`get_perct_resp_t()` `-a perct-respt`

`get_tran_avg()` `-a tran`

`get_tran_cdf_resp_t()` `-a tran-cdf-respt`

`get_tran_cdf_pass_t()` `-a tran-cdf-passt`

`get_tran_prob()` `-a tranprob`

`get_tran_reward()` `-a tranreward`

`get_tran_var()` `-a tranvar`

`get_prob()` `-a prob` (station, aggregate and system)

`get_prob_marg()` `-a marg`

`get_norm_const()` `-a normconst`

`get_generator()` `-a gen`

`get_state_space()` `-a states`

`get_state_vec()` `-a statevec`

`sample()` `-a sample` (all four samplers)

`get_reward()` `-a reward`

`get_reward_value()` `-a reward-value`

`get_sens()` `-a sens`

`get_first_pass_t()` `-a first-passt`

`get_var()` `-a var`

`get_avg_busy_period()` `-a busyperiod`

`get_odes()` `-a odes`

`get_jacobian()` `-a jacobian`

`get_aoi()` `-a aoi`

`get_internals()` `-a internals`

`get_bounds()` `-a bounds`

`get_posterior()` `-a posterior`

`get_interval()` `-a interval`

`get_cache_avg_table()` `-a cache`

`get_item_avg_table()` `-a item`

`get_avg_fj_table()` `-a fj`

`get_stage_table()` `-a stage`

Public fields

`model` The model this solver was built on.

`name` The solver's own name, e.g. `"MVA"`.

`options` The knobs this solver was built with.

Methods

Public methods:

- `NetworkSolver$new()`
- `NetworkSolver$reset()`
- `NetworkSolver$is_solved()`
- `NetworkSolver$avg_table()`
- `NetworkSolver$avg_sys_table()`
- `NetworkSolver$avg_node_table()`
- `NetworkSolver$avg_chain_table()`
- `NetworkSolver$run_analyzer()`
- `NetworkSolver$analysis()`
- `NetworkSolver$get_avg()`
- `NetworkSolver$method_used()`
- `NetworkSolver$list_valid_methods()`
- `NetworkSolver$print()`
- `NetworkSolver$get_avg_node_chain_table()`
- `NetworkSolver$get_cdf_resp_t()`
- `NetworkSolver$get_cdf_pass_t()`
- `NetworkSolver$get_perct_resp_t()`
- `NetworkSolver$get_tran_avg()`
- `NetworkSolver$get_tran_cdf_resp_t()`
- `NetworkSolver$get_tran_cdf_pass_t()`
- `NetworkSolver$get_tran_prob()`
- `NetworkSolver$get_tran_reward()`
- `NetworkSolver$get_tran_var()`
- `NetworkSolver$get_prob()`
- `NetworkSolver$get_prob_marg()`
- `NetworkSolver$get_norm_const()`
- `NetworkSolver$get_generator()`
- `NetworkSolver$get_state_space()`
- `NetworkSolver$get_state_vec()`
- `NetworkSolver$sample()`
- `NetworkSolver$get_reward()`
- `NetworkSolver$get_reward_value()`
- `NetworkSolver$get_sens()`
- `NetworkSolver$get_first_pass_t()`

- NetworkSolver$get_var()
- NetworkSolver$get_avg_busy_period()
- NetworkSolver$get_odes()
- NetworkSolver$get_jacobian()
- NetworkSolver$get_aoi()
- NetworkSolver$get_internals()
- NetworkSolver$get_bounds()
- NetworkSolver$get_posterior()
- NetworkSolver$get_interval()
- NetworkSolver$get_cache_avg_table()
- NetworkSolver$get_item_avg_table()
- NetworkSolver$get_avg_fj_table()
- NetworkSolver$get_stage_table()

`NetworkSolver$new()`: Construct a solver over a model.

Usage:

```
NetworkSolver$new(model, method = NULL, ...)
```

Arguments:

`model` A `Network`.

`method` Optional method name within the solver.

`...` Further CLI knobs, e.g. `samples = 1e5`, `seed = 23000`, `tol = 1e-8`, `arith = "rational"`. A name is spelled as the flag without its dashes and with underscores for dashes, so `iter_max` is `--iter_max` and `max_states` is `--max-states`.

`NetworkSolver$reset()`: Discard every cached solve.

Usage:

```
NetworkSolver$reset()
```

`NetworkSolver$is_solved()`: Whether any analysis has been solved yet.

Usage:

```
NetworkSolver$is_solved()
```

`NetworkSolver$avg_table()`: The per-station average table.

Usage:

```
NetworkSolver$avg_table()
```

`NetworkSolver$avg_sys_table()`: The per-chain system table.

Usage:

```
NetworkSolver$avg_sys_table()
```

`NetworkSolver$avg_node_table()`: The per-node table.

Usage:

```
NetworkSolver$avg_node_table()
```

`NetworkSolver$avg_chain_table()`: The per-chain station table.

Usage:

```
NetworkSolver$avg_chain_table()
```

`NetworkSolver$run_analyzer()`: Force the solve and return the average table.

Usage:

```
NetworkSolver$run_analyzer()
```

`NetworkSolver$analysis()`: Any analysis the CLI serves, by its `-a` name.

Usage:

```
NetworkSolver$analysis(name, ..., raw = FALSE)
```

Arguments:

`name` The analysis, e.g. `"cdf-respt"`, `"tranprob"`.

`...` Knobs for this analysis only, merged over the solver's.

`raw` When `TRUE`, return the parsed JSON payload untouched rather than a table. Analyses that are not tabular (a generator, a state space, a set of CDF curves) have no data.frame shape, and pretending otherwise would flatten a structure the caller needs.

`NetworkSolver$get_avg()`: The six metrics as (station x class) matrices.

Built from the model's own station and class lists and scattered by name, because the table is not a complete grid: the engine drops any row whose metrics are all exactly zero, so its dimensions do not imply the model's.

Usage:

```
NetworkSolver$get_avg()
```

`NetworkSolver$method_used()`: The method the solve actually resolved to.

Usage:

```
NetworkSolver$method_used()
```

`NetworkSolver$list_valid_methods()`: The methods this solver advertises on this model.

Usage:

```
NetworkSolver$list_valid_methods()
```

`NetworkSolver$print()`: Print a one-line description.

Usage:

```
NetworkSolver$print(...)
```

Arguments:

`...` Ignored.

`NetworkSolver$get_avg_node_chain_table()`:

Usage:

```
NetworkSolver$get_avg_node_chain_table(..., raw = FALSE)
```

`NetworkSolver$get_cdf_resp_t()`:

Usage:

```
NetworkSolver$get_cdf_resp_t(..., raw = TRUE)
```

`NetworkSolver$get_cdf_pass_t()`:

Usage:

```
NetworkSolver$get_cdf_pass_t(..., raw = TRUE)
```

`NetworkSolver$get_perct_resp_t()`:

Usage:

```
NetworkSolver$get_perct_resp_t(..., raw = TRUE)
```

`NetworkSolver$get_tran_avg()`:

Usage:

```
NetworkSolver$get_tran_avg(..., raw = TRUE)
```

`NetworkSolver$get_tran_cdf_resp_t()`:

Usage:

```
NetworkSolver$get_tran_cdf_resp_t(..., raw = TRUE)
```

`NetworkSolver$get_tran_cdf_pass_t()`:

Usage:

```
NetworkSolver$get_tran_cdf_pass_t(..., raw = TRUE)
```

`NetworkSolver$get_tran_prob()`:

Usage:

```
NetworkSolver$get_tran_prob(..., raw = TRUE)
```

`NetworkSolver$get_tran_reward()`:

Usage:

```
NetworkSolver$get_tran_reward(..., raw = TRUE)
```

`NetworkSolver$get_tran_var()`:

Usage:

```
NetworkSolver$get_tran_var(..., raw = TRUE)
```

`NetworkSolver$get_prob()`:

Usage:

```
NetworkSolver$get_prob(..., raw = TRUE)
```

`NetworkSolver$get_prob_marg()`:

Usage:

```
NetworkSolver$get_prob_marg(..., raw = TRUE)
```

`NetworkSolver$get_norm_const()`:

Usage:

```
NetworkSolver$get_norm_const(..., raw = TRUE)
```

`NetworkSolver$get_generator()`:

Usage:

```
NetworkSolver$get_generator(..., raw = TRUE)
```

`NetworkSolver$get_state_space()`:

Usage:

```
NetworkSolver$get_state_space(..., raw = TRUE)
```

`NetworkSolver$get_state_vec()`:

Usage:

```
NetworkSolver$get_state_vec(..., raw = TRUE)
```

`NetworkSolver$sample()`:

Usage:

```
NetworkSolver$sample(..., raw = TRUE)
```

`NetworkSolver$get_reward()`:

Usage:

```
NetworkSolver$get_reward(..., raw = TRUE)
```

`NetworkSolver$get_reward_value()`:

Usage:

```
NetworkSolver$get_reward_value(..., raw = TRUE)
```

`NetworkSolver$get_sens()`:

Usage:

```
NetworkSolver$get_sens(..., raw = TRUE)
```

`NetworkSolver$get_first_pass_t()`:

Usage:

```
NetworkSolver$get_first_pass_t(..., raw = TRUE)
```

`NetworkSolver$get_var()`:

Usage:

```
NetworkSolver$get_var(..., raw = TRUE)
```

`NetworkSolver$get_avg_busy_period()`:

Usage:

```
NetworkSolver$get_avg_busy_period(..., raw = TRUE)
```

`NetworkSolver$get_odes()`:

Usage:

```
NetworkSolver$get_odes(..., raw = TRUE)
```

`NetworkSolver$get_jacobian()`:

Usage:

```
NetworkSolver$get_jacobian(..., raw = TRUE)
```

`NetworkSolver$get_aoi()`:

Usage:

```
NetworkSolver$get_aoi(..., raw = TRUE)
```

`NetworkSolver$get_internals()`:

Usage:

```
NetworkSolver$get_internals(..., raw = TRUE)
```

`NetworkSolver$get_bounds()`:

Usage:

```
NetworkSolver$get_bounds(..., raw = TRUE)
```

`NetworkSolver$get_posterior()`:

Usage:

```
NetworkSolver$get_posterior(..., raw = TRUE)
```

`NetworkSolver$get_interval()`:

Usage:

```
NetworkSolver$get_interval(..., raw = TRUE)
```

`NetworkSolver$get_cache_avg_table()`:

Usage:

```
NetworkSolver$get_cache_avg_table(..., raw = FALSE)
```

`NetworkSolver$get_item_avg_table()`:

Usage:

```
NetworkSolver$get_item_avg_table(..., raw = FALSE)
```

`NetworkSolver$get_avg_fj_table()`:

Usage:

```
NetworkSolver$get_avg_fj_table(..., raw = FALSE)
```

`NetworkSolver$get_stage_table()`:

Usage:

```
NetworkSolver$get_stage_table(..., raw = FALSE)
```

OpenClass *An open job class*

Description

An open job class

Super class

JobClass -> OpenClass

Methods

Public methods:

- OpenClass$new()
- OpenClass$clone()

OpenClass$new(): Create an open class.

Usage:

```
OpenClass$new(model, name, prio = 0)
```

Arguments:

model Owning model.

name Class name.

prio Priority; 0 is the default.

OpenClass$clone(): The objects of this class are cloneable with this method.

Usage:

```
OpenClass$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Place *Petri-net place*

Description

A place of a stochastic Petri net. It holds tokens of each job class, or, when declared FLUID with 'set_fluid()', one real-valued level instead: the fluid place of a fluid stochastic Petri net (FSPN). A fluid place is moved only by the fluid arcs of a 'Transition' mode ('set_fluid_rate()', 'set_fluid_impulse()') and read by its guards and rate dependences.

A place given a scheduling strategy is a QUEUEING place, and takes service and servers as a 'Queue' does.

Super classes

LineNode -> ServiceNode -> Place

Methods

Public methods:

- Place$new()
- Place$set_initial_state()
- Place$set_departure_discipline()
- Place$set_fluid()
- Place$set_fluid_capacity()
- Place$set_fluid_level()
- Place$clone()

Place$new(): Create a place.

Usage:

```
Place$new(model, name, sched_strategy = NULL)
```

Arguments:

model Owning model.

name Node name.

sched_strategy Optional SchedStrategy member, for a queueing place.

Place$set_initial_state(): Initial marking: the token count of each job class, in the model's class order. Emitted as the node's 'initialState'.

Usage:

```
Place$set_initial_state(x)
```

Arguments:

x Numeric vector of token counts.

Place$set_departure_discipline(): Order in which the place releases tokens of one class.

Usage:

```
Place$set_departure_discipline(cls, discipline)
```

Arguments:

cls Job class.

discipline A DepartureDiscipline member.

Place$set_fluid(): Declare the place FLUID: it holds one real level in [0, fluid capacity] instead of a token count per class. Token arcs into or out of a fluid place are refused by the engine.

Usage:

```
Place$set_fluid(on = TRUE)
```

Arguments:

on Whether the place is fluid; TRUE by default. FALSE also drops the fluid capacity and level, which mean nothing on a token place.

`Place$set_fluid_capacity()`: Upper bound B of a fluid place's level. Inflow that would push the level past B is lost. 'Inf' or NULL makes the place unbounded, which is what an omitted 'fluidCapacity' means on the wire.

Usage:

```
Place$set_fluid_capacity(capacity)
```

Arguments:

`capacity` A positive number, 'Inf', or NULL.

`Place$set_fluid_level()`: Initial level of a fluid place. Omitted means 0.

Usage:

```
Place$set_fluid_level(level)
```

Arguments:

`level` A non-negative number.

`Place$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Place$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`Queue` *Queueing station*

Description

A Queue with 'SchedStrategy$INF' IS a delay station in every codebase: the server count becomes infinite and the struct reports it as a Delay. Use 'Delay' when that is what you mean.

Super classes

`LineNode` -> `ServiceNode` -> `Queue`

Methods

Public methods:

- `Queue$new()`
- `Queue$clone()`

`Queue$new()`: Create a queue.

Usage:

```
Queue$new(model, name, sched_strategy = SchedStrategy$FCFS)
```

Arguments:

`model` Owning model.

`name` Node name.

`sched_strategy` A SchedStrategy member; defaults to FCFS.

`Queue$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Queue$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`Router` *Router (a node that only routes)*

Description

Router (a node that only routes)

Super class

`LineNode` -> `Router`

Methods

Public methods:

- `Router$new()`
- `Router$clone()`

`Router$new()`: Create a router.

Usage:

```
Router$new(model, name)
```

Arguments:

`model` Owning model.

`name` Node name.

`Router$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Router$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`RoutingMatrix` *Routing matrix*

Description

Build with ‘model$init_routing_matrix()‘, then ‘$set(...)‘. Entries are addressed by object or by name; probabilities that are never set are zero.

Methods

Public methods:

- `RoutingMatrix$new()`
- `RoutingMatrix$set()`
- `RoutingMatrix$get()`
- `RoutingMatrix$.entries()`
- `RoutingMatrix$.model_name()`
- `RoutingMatrix$print()`
- `RoutingMatrix$clone()`

`RoutingMatrix$new()`: Create an empty routing matrix for a model.

Usage:

```
RoutingMatrix$new(model)
```

Arguments:

`model` The model whose nodes and classes it addresses.

`RoutingMatrix$set()`: Set one routing probability.
Two arities, matching the other codebases: ‘$set(fromClass, toClass, fromNode, toNode, p)‘ sets one class-pair entry, and ‘$set(class, fromNode, toNode, p)‘ sets it within a single class.

Usage:

```
RoutingMatrix$set(...)
```

Arguments:

`...` See above.

`RoutingMatrix$get()`: The probability at one entry, or 0 when unset.

Usage:

```
RoutingMatrix$get(fromClass, toClass, fromNode, toNode)
```

Arguments:

`fromClass, toClass` Job classes.

`fromNode, toNode` Nodes.

`RoutingMatrix$.entries()`: The accumulated entries, as the wire nests them.

Usage:

```
RoutingMatrix$.entries()
```

`RoutingMatrix$.model_name()`: The owning model's name, used to refuse a cross-model link.

Usage:

```
RoutingMatrix$.model_name()
```

`RoutingMatrix$print()`: Print a summary.

Usage:

```
RoutingMatrix$print(...)
```

Arguments:

`...` Ignored.

`RoutingMatrix$clone()`: The objects of this class are cloneable with this method.

Usage:

```
RoutingMatrix$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`save_model` *Write a model as a 'line-model' document*

Description

The portable JSON interchange shared with the MATLAB, Java, Python and C++ editions, specified by 'doc/line-model.schema.json'.

Usage

```
save_model(model, filename = NULL, pretty = TRUE)
```

Arguments

`model`	A 'Network'.
`filename`	Path to write; when NULL the JSON is returned as a string.
`pretty`	Indent the output.

Value

Invisibly the JSON string.

sc *Mark a value as a JSON scalar rather than a one-element array*

Description

Mark a value as a JSON scalar rather than a one-element array

Usage

```
sc(x)
```

Arguments

x	A length-1 value.

SelfLoopingClass *A self-looping job class*

Description

A self-looping job class

Super class

JobClass -> SelfLoopingClass

Methods

Public methods:

- SelfLoopingClass$new()
- SelfLoopingClass$clone()

SelfLoopingClass$new(): Create a self-looping class.

Usage:

```
SelfLoopingClass$new(model, name, njobs, refstat, prio = 0)
```

Arguments:

model Owning model.

name Class name.

njobs Population.

refstat Reference station.

prio Priority.

SelfLoopingClass$clone(): The objects of this class are cloneable with this method.

Usage:

```
SelfLoopingClass$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`serial_routing` *Route jobs through a sequence of nodes*

Description

A topology CONSTRUCTOR, mirroring ‘Network.serial_routing‘ in the other codebases: it chains the nodes in order, and if the last node is not a Sink it closes the cycle back to the first. This is deliberately the one piece of LINE topology semantics that lives in R, because it shapes a document rather than deriving a quantity.

Usage

```
serial_routing(...)
```

Arguments

`...`	Nodes, in visiting order.

Details

Applies to every class in the model. For per-class chains, build a ‘RoutingMatrix‘ and call ‘$set()‘.

Value

A ‘RoutingMatrix‘.

`SignalClass` *A signal class (G-network)*

Description

A signal arriving at a station removes jobs (negative), empties it (catastrophe) or answers a request (reply) instead of joining it.

Super class

`JobClass` -> `SignalClass`

Methods

Public methods:

- SignalClass$new()
- SignalClass$clone()

SignalClass$new(): Create a signal class.

Usage:

```
SignalClass$new(
  model,
  name,
  signal_type = "negative",
  target_class = NULL,
  removal_distribution = NULL,
  removal_policy = "random",
  refstat = NULL,
  prio = 0
)
```

Arguments:

model Owning model.

name Class name.

signal_type ‘"negative"‘, ‘"reply"‘ or ‘"catastrophe"‘.

target_class Class whose jobs the signal acts on (or NULL).

removal_distribution Distribution of the number of jobs one negative signal removes (default one).

removal_policy ‘"random"‘, ‘"fcfs"‘ or ‘"lcfs"‘.

refstat For a CLOSED signal, its reference station; NULL for open.

prio Priority.

SignalClass$clone(): The objects of this class are cloneable with this method.

Usage:

```
SignalClass$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Sink *Sink*

Description

Sink

Super class

LineNode -> Sink

Methods

Public methods:

- `Sink$new()`
- `Sink$clone()`

`Sink$new()`: Create a sink.

Usage:

```
Sink$new(model, name)
```

Arguments:

`model` Owning model.

`name` Node name.

`Sink$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Sink$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`Solvers` *Solvers*

Description

Each is constructed as `MVA$new(model)` or `MVA$new(model, "method")`, mirroring the Python and C++ spelling.

Super class

`NetworkSolver` -> `SolverMVA`

Super class

`NetworkSolver` -> `SolverNC`

Super class

`NetworkSolver` -> `SolverCTMC`

Super class

`NetworkSolver` -> `SolverFLD`

Super class

`NetworkSolver` -> `SolverMAM`

Super class

NetworkSolver -> SolverSSA

Super class

NetworkSolver -> SolverBA

Super class

NetworkSolver -> SolverAUTO

Super class

NetworkSolver -> SolverLDES

Super class

NetworkSolver -> SolverJMT

Super class

NetworkSolver -> SolverLN

Super class

NetworkSolver -> SolverENV

Super class

NetworkSolver -> SolverAG

Super class

NetworkSolver -> SolverLQNS

Super class

NetworkSolver -> SolverUQ

Super class

NetworkSolver -> SolverMVA

Super class

NetworkSolver -> SolverNC

Super class

NetworkSolver -> SolverCTMC

Super class

NetworkSolver -> SolverFLD

Super class

NetworkSolver -> SolverMAM

Super class

NetworkSolver -> SolverSSA

Super class

NetworkSolver -> SolverAUTO

Super class

NetworkSolver -> SolverLDES

Super class

NetworkSolver -> SolverJMT

Super class

NetworkSolver -> SolverLN

Super class

NetworkSolver -> SolverENV

Super class

NetworkSolver -> SolverBA

Super class

NetworkSolver -> SolverAG

Super class

NetworkSolver -> SolverLQNS

Super class

NetworkSolver -> SolverUQ

Super class

NetworkSolver -> SolverFLD

solver_meta *The provenance of a result table*

Description

The resolved method, the arithmetic, the iteration count and any warning the engine raised, carried as an attribute rather than as a column because they describe the solve and not a row.

Usage

```
solver_meta(x)
```

Arguments

x	A table returned by a solver.

Value

A list with `method`, `arith`, `iter`, `converged`, `warning` and `timed_out`.

Source *Arrival source*

Description

Arrival source

Super class

`LineNode` -> `Source`

Methods

Public methods:

- `Source$new()`
- `Source$set_arrival()`
- `Source$set_marked_arrival()`
- `Source$set_arrival_batch()`
- `Source$set_service()`
- `Source$clone()`

`Source$new()`: Create a source.

Usage:

```
Source$new(model, name)
```

Arguments:

`model` Owning model.

`name` Node name.

`Source$set_arrival()`: Set the arrival process for one open class.

Usage:

```
Source$set_arrival(cls, dist)
```

Arguments:

`cls` Job class.

`dist` A distribution.

`Source$set_marked_arrival()`: Bind a MARKED arrival process to one class per mark.

A mark is a label the process attaches to each event, and at a Source it selects the CLASS of the arriving job: mark k emits jobs of 'classes[[k]]', with all marks driven by ONE shared modulating chain. That sharing is the point, and it is why this is not the same as calling 'set_arrival' per class with copies of the process: K copies are K independent streams.

Takes any marked family the engine declares: 'MPH' (stationary), 'MMAPt' and 'MPHt' (schedule-bearing), or 'BMMAPt', whose epochs also release a BATCH of jobs that all carry the mark's class.

Usage:

```
Source$set_marked_arrival(dist, classes)
```

Arguments:

`dist` A marked distribution.

`classes` A list of K distinct open classes, ordered by mark.

`Source$set_arrival_batch()`: Release a BATCH of jobs at each arrival epoch of one class.

The interarrival law set by 'set_arrival' keeps spacing the epochs; this says how many jobs each one releases. Mutually exclusive with a process that carries its own batch sizes ('BMAP', 'BMMAPt'), which the engine refuses by name.

Usage:

```
Source$set_arrival_batch(cls, dist)
```

Arguments:

`cls` Job class.

`dist` A discrete distribution supported on {1,2,...}.

`Source$set_service()`: Refused: a Source has arrivals, not service.

Usage:

```
Source$set_service(...)
```

Arguments:

`...` Ignored.

`Source$clone()`: The objects of this class are cloneable with this method.

Usage:

```
Source$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

tget *One cell of a result table, by station and class name*

Description

The engine drops a (station, class) row whose metrics are all exactly zero, so a name pair that is absent is a zero rather than a gap. This returns 0 for such a pair and NA only for a column the solver did not compute.

Usage

```
tget(tbl, column, station, jobclass = NULL)
```

Arguments

tbl	A table returned by a solver.
column	Column name, e.g. "QLen".
station	Station name.
jobclass	Class name.

Transition *Petri-net transition*

Description

A transition of a stochastic Petri net, made of one or more firing MODES. A mode is created with 'add_mode()', which returns its name, and every other setter takes that name (or the mode's 1-based index) as its first argument.

Besides the token arcs ('set_enabling_conditions()', 'set_inhibiting_conditions()', 'set_firing_outcome()'), a mode may carry the FLUID arcs of a fluid stochastic Petri net, each naming a fluid 'Place': a flow while the mode runs ('set_fluid_rate()'), a level guard ('set_fluid_guard()'), a jump at each firing ('set_fluid_impulse()'), a level-dependent multiplier of its firing rate ('set_fluid_rate_dependence()'), and a marking-dependent multiplier of a flow ('set_fluid_flow_dependence()'). A timed exponential mode may also scale its own firing rate by a function of the marking ('set_firing_rate_dependence()').

Super class

LineNode -> Transition

Methods

Public methods:

- Transition$new()
- Transition$add_mode()
- Transition$get_modes()
- Transition$set_distribution()
- Transition$set_timing_strategy()
- Transition$set_number_of_servers()
- Transition$set_firing_priorities()
- Transition$set_firing_weights()
- Transition$set_firing_rate_dependence()
- Transition$set_enabling_conditions()
- Transition$set_inhibiting_conditions()
- Transition$set_firing_outcome()
- Transition$set_fluid_rate()
- Transition$set_fluid_guard()
- Transition$set_fluid_impulse()
- Transition$set_fluid_rate_dependence()
- Transition$set_fluid_flow_dependence()
- Transition$clone()

`Transition$new()`: Create a transition with no modes.

Usage:

```
Transition$new(model, name)
```

Arguments:

`model` Owning model.

`name` Node name.

`Transition$add_mode()`: Add a firing mode.

Usage:

```
Transition$add_mode(name)
```

Arguments:

`name` Mode name, unique within this transition.

Returns: The mode name, which the other setters take.

`Transition$get_modes()`: Names of the modes, in the order they were added.

Usage:

```
Transition$get_modes()
```

`Transition$set_distribution()`: Firing-time distribution of a mode.

Usage:

```
Transition$set_distribution(mode, dist)
```

Arguments:

`mode` Mode name or index.

`dist` A distribution.

`Transition$set_timing_strategy()`: Whether a mode fires after a delay or at once.

Usage:

```
Transition$set_timing_strategy(mode, strategy)
```

Arguments:

`mode` Mode name or index.

`strategy` A TimingStrategy member.

`Transition$set_number_of_servers()`: Number of firing servers of a mode; 'Inf' for an infinite-server mode.

Usage:

```
Transition$set_number_of_servers(mode, n)
```

Arguments:

`mode` Mode name or index.

`n` Server count, or 'Inf'.

`Transition$set_firing_priorities()`: Priority among simultaneously enabled modes; higher fires first.

Usage:

```
Transition$set_firing_priorities(mode, priority)
```

Arguments:

`mode` Mode name or index.

`priority` Integer priority.

`Transition$set_firing_weights()`: Weight randomising the choice among enabled modes of equal highest priority.

Usage:

```
Transition$set_firing_weights(mode, weight)
```

Arguments:

`mode` Mode name or index.

`weight` Non-negative weight.

`Transition$set_firing_rate_dependence()`: Multiply the firing rate of an exponential TIMED mode by a function g(M) of the marking: an enabled binding fires at rate * g(M). Exact only for memoryless firing, so an IMMEDIATE mode or a non-exponential mode is refused.

'g' receives the marking matrix M, one row per node of the model in the order the nodes were created and one column per class, named by node and class. M holds the (place, class) slots of the mode's enabling conditions and is 0 on every other entry.

The document cannot carry a function, so 'save_model()' tabulates g over the box lattice of the enabling slots (node-major, class-minor, slot 0 fastest), each cut off at the place capacity, or at 10 tokens on an unbounded place; a marking outside the table is read as 1. The lattice is built when the document is written, so it sees the enabling conditions and capacities set by then.

Usage:

```
Transition$set_firing_rate_dependence(mode, g)
```

Arguments:

`mode` Mode name or index.

`g` A function of the marking matrix returning a positive number; or a table with 'slots', 'cutoffs' and 'scaling' as the document writes it; or NULL to restore the unit multiplier.

`Transition$set_enabling_conditions()`: Tokens of one class a place must hold for the mode to be enabled. Setting a pair again replaces it; 0 removes it.

Usage:

```
Transition$set_enabling_conditions(mode, cls, place, count)
```

Arguments:

`mode` Mode name or index.

`cls` Job class.

`place` Input Place.

`count` Token multiplicity.

`Transition$set_inhibiting_conditions()`: Tokens of one class on a place above which the mode is inhibited. Setting a pair again replaces it; 'Inf' removes it.

Usage:

```
Transition$set_inhibiting_conditions(mode, cls, place, count)
```

Arguments:

`mode` Mode name or index.

`cls` Job class.

`place` Input Place.

`count` Token multiplicity.

`Transition$set_firing_outcome()`: Tokens of one class deposited on a place when the mode fires. Setting a pair again replaces it; 0 removes it.

Usage:

```
Transition$set_firing_outcome(mode, cls, place, count)
```

Arguments:

`mode` Mode name or index.

`cls` Job class.

`place` Output node: a Place, or e.g. the Sink of an open net.

`count` Token multiplicity.

`Transition$set_fluid_rate()`: Fluid flow of a TIMED mode into ('rate > 0') or out of ('rate < 0') a fluid place, per running server: while k servers of the mode run, the level moves at k*rate. A flow stops at the place's bounds. 0 removes the flow.

Usage:

```
Transition$set_fluid_rate(mode, place, rate)
```

Arguments:

`mode` Mode name or index.

`place` A fluid Place.

`rate` Level change per unit time per running server.

`Transition$set_fluid_guard()`: Enable the mode only while the level of a fluid place satisfies 'level op threshold', inclusive at the threshold. One guard per place and mode; all of them must hold. 'op = NULL' removes it.

Usage:

```
Transition$set_fluid_guard(mode, place, op, threshold = NULL)
```

Arguments:

`mode` Mode name or index.

`place` A fluid Place.

`op` '">="' or '"<="', or NULL to remove the guard.

`threshold` Level threshold.

`Transition$set_fluid_impulse()`: Move the level of a fluid place by 'delta' at every firing of the mode, clipped at the place's bounds. 0 removes the impulse.

Usage:

```
Transition$set_fluid_impulse(mode, place, delta)
```

Arguments:

`mode` Mode name or index.

`place` A fluid Place.

`delta` Level jump at each firing.

`Transition$set_fluid_rate_dependence()`: Multiply the firing rate of an exponential TIMED mode by h(level) of a fluid place. Several places multiply.

The document cannot carry a function, so it carries a TABLE: the multiplier is the piecewise-linear interpolant of (levels, values), held constant beyond the last level. Supply the table directly, or an R function, which 'save_model()' samples on 201 equally spaced levels over [0, L]: L is the place's fluid capacity when finite, otherwise max(10, 10 * max(1, initial level, |guard thresholds on the place|)), the rule every other edition applies. A function is sampled when the document is written, so it sees the capacity and guards set by then.

Usage:

```
Transition$set_fluid_rate_dependence(mode, place, h)
```

Arguments:

`mode` Mode name or index.

`place` A fluid Place.

`h` A function of the level returning a non-negative number; or a list or data frame with numeric 'levels' (strictly increasing, from 0) and 'values'; or NULL to remove the dependence.

`Transition$set_fluid_flow_dependence()`: Scale the flow of a mode into or out of a fluid place by a function of the MARKING: while k servers run, the level moves at k * rate * phi(M), 'rate' being the one set by 'set_fluid_rate()'.

'phi' receives the marking matrix M, one row per node of the model in the order the nodes were created and one column per class, named by node and class, so 'M["On", "Source"]' reads the

tokens of class Source on place On. M holds the token places ‘read‘ and is 0 on every other row. Without ‘read‘, M holds the input places of the mode, resolved when the document is written.

The document cannot carry a function, so ‘save_model()‘ tabulates phi over the box lattice of the (read place, class) counts, each cut off at the place capacity, at the population of a closed class, and at 10 tokens for an open class; a marking outside the table is read as 1.

Usage:

```
Transition$set_fluid_flow_dependence(mode, place, phi, read = NULL)
```

Arguments:

`mode` Mode name or index.

`place` A fluid Place the mode has a flow on.

`phi` A function of the marking matrix returning a non-negative number; or a table with ‘slots‘, ‘cutoffs‘ and ‘scaling‘ as the document writes it; or NULL to remove the dependence.

`read` The token Places phi reads (nodes or names), or NULL for the input places of the mode.

`Transition$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Transition$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`validate_model` *Check a model against the engine's reader without solving it*

Description

The reader refuses by name: an unknown scheduling strategy, a distribution whose parameters do not fit, a routing matrix whose rows do not sum. This is that refusal with no solve behind it, so it is cheap enough to call in a documentation example.

Usage

```
validate_model(model)
```

Arguments

`model`	A `Network`, or a model document as a string.

Value

Invisibly `TRUE`. Raises with the reader's own message otherwise.

Index

AG *(Solvers)*, 52
APH *(Distributions)*, 10
api_invoke, *3*, 3
api_list, 3
AUTO *(Solvers)*, 52
AvgTable, 4

BA *(Solvers)*, 52
Bernoulli *(Distributions)*, 10
Binomial *(Distributions)*, 10
BMAP *(Distributions)*, 10
BMMAPt *(Distributions)*, 10

Cache, 4
capi_available, 6
capi_run, 7
capi_status, 7
ClassSwitch, 8
ClosedClass, 9
Cox2 *(Distributions)*, 10
Coxian *(Distributions)*, 10
CTMC *(Solvers)*, 52

Delay, 10
DepartureDiscipline *(enums)*, 13
Det *(Distributions)*, 10
Disabled *(Distributions)*, 10
DiscreteSampler *(Distributions)*, 10
DiscreteUniform *(Distributions)*, 10
Distributions, 10
DMAP *(Distributions)*, 10
DropRule *(enums)*, 13

EmpiricalCDF *(Distributions)*, 10
enums, 13
ENV *(Solvers)*, 52
Erlang *(Distributions)*, 10
Exp *(Distributions)*, 10

find_solver, 14
FLD *(Solvers)*, 52

Fork, 14

Gamma *(Distributions)*, 10
Geometric *(Distributions)*, 10

HyperExp *(Distributions)*, 10

Immediate *(Distributions)*, 10
ImpatienceType *(enums)*, 13

JMT *(Solvers)*, 52
JobClass, *9*, 15, *43*, *49*, *50*
Join, 17

LayeredNetwork, 18
LayeredNetworkGenerator, 20
LDES *(Solvers)*, 52
line_version, 26
LineNode, *4*, *8*, *10*, *14*, *17*, 23, *27*, *44–46*, *51*, *55*, *57*
LN *(Solvers)*, 52
load_model, 26
Logger, 27
Lognormal *(Distributions)*, 10
LQNS *(Solvers)*, 52

MAM *(Solvers)*, 52
MAP *(Distributions)*, 10
MAPt *(Distributions)*, 10
MarkedMMPP *(Distributions)*, 10
ME *(Distributions)*, 10
MMAP *(Distributions)*, 10
MMAPt *(Distributions)*, 10
MMPP2 *(Distributions)*, 10
model_index, 27
MPH *(Distributions)*, 10
MPHt *(Distributions)*, 10
MVA *(Solvers)*, 52

NC *(Solvers)*, 52
Network, 28

NetworkGenerator, 32
NetworkSolver, 35, *52–54*
NHPP (Distributions), 10
Normal (Distributions), 10

OpenClass, 43

Pareto (Distributions), 10
PH (Distributions), 10
PHt (Distributions), 10
Place, 43
Poisson (Distributions), 10
Prior (Distributions), 10
PriorContinuous (Distributions), 10

Queue, 45

RAP (Distributions), 10
ReplacementStrategy (enums), 13
Replayer (Distributions), 10
Router, 46
RoutingMatrix, 47
RoutingStrategy (enums), 13

save_model, 48
sc, 49
SchedStrategy (enums), 13
SelfLoopingClass, 49
serial_routing, 50
ServiceNode, *10*, *44*, *45*
SignalClass, 50
Sink, 51
solver_meta, 55
SolverAG (Solvers), 52
SolverAUTO (Solvers), 52
SolverBA (Solvers), 52
SolverCTMC (Solvers), 52
SolverENV (Solvers), 52
SolverFLD (Solvers), 52
SolverFluid (Solvers), 52
SolverJMT (Solvers), 52
SolverLDES (Solvers), 52
SolverLN (Solvers), 52
SolverLQNS (Solvers), 52
SolverMAM (Solvers), 52
SolverMVA (Solvers), 52
SolverNC (Solvers), 52
Solvers, 52
SolverSSA (Solvers), 52
SolverUQ (Solvers), 52
Source, 55
SSA (Solvers), 52

tget, 57
TimingStrategy (enums), 13
Transition, 57

Uniform (Distributions), 10
UQ (Solvers), 52

validate_model, 62

Weibull (Distributions), 10

Zipf (Distributions), 10