Package 'sdbuildR'

May 9, 2026

Title Easily Build, Simulate, and Visualise Stock-and-Flow Models

Version 1.0.8

Date 2025-10-23

Description Stock-and-flow models are a computational method from the field of system dynamics. They represent how systems change over time and are mathematically equivalent to ordinary differential equations. 'sdbuildR' (system dynamics builder) provides an intuitive interface for constructing stock-and-flow models without requiring extensive domain knowledge. Models can quickly be simulated and revised, supporting iterative development. 'sdbuildR' simulates models in 'R' and 'Julia', where 'Julia' offers unit support and large-scale ensemble simulations. Additionally, 'sdbuildR' can import models created in 'Insight Maker' (<https://insightmaker.com/>).

URL https://kcevers.github.io/sdbuildR/, https://github.com/KCEvers/sdbuildR

BugReports https://github.com/KCEvers/sdbuildR/issues

Depends R (>= 4.2.0)

Imports data.table, deSolve, DiagrammeR, dplyr, igraph, jsonlite, JuliaConnectoR, magrittr, parallel, plotly, purrr, rlang, rvest, stringr, xml2

Suggests DiagrammeRsvg, htmlwidgets, kableExtra, knitr, rsvg, styler, testthat (>= 3.0.0), textutils, webshot2

Config/Needs/website rmarkdown

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2

License GPL (>= 3)

NeedsCompilation no

Author Kyra Caitlin Evers [aut, cre, cph] (ORCID: <https://orcid.org/0000-0001-6890-3482>)

Maintainer Kyra Caitlin Evers <kyra.c.evers@gmail.com>

Repository CRAN

Date/Publication 2025-11-19 12:20:02 UTC

Contents

`as.data.frame.sdbuildR_sim` *Create data frame of simulation results*

Description

Convert simulation results to a data.frame.

Usage

```
## S3 method for class 'sdbuildR_sim'
as.data.frame(x, row.names = NULL, optional = FALSE, direction = "long", ...)
```

Arguments

`x`	Output of simulate().
`row.names`	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
`optional`	Ignored parameter.
`direction`	Format of data frame, either "long" (default) or "wide".
`...`	Optional parameters

Value

A data.frame with simulation results. For `direction = "long"` (default), the data frame has three columns: `time`, `variable`, and `value`. For `direction = "wide"`, the data frame has columns `time` followed by one column per variable.

See Also

`simulate()`, `xmile()`

Examples

```
sfm <- xmile("SIR")
sim <- simulate(sfm)
df <- as.data.frame(sim)
head(df)

# Get results in wide format
```

```
df_wide <- as.data.frame(sim, direction = "wide")
head(df_wide)
```

`as.data.frame.sdbuildR_xmile` *Convert stock-and-flow model to data frame*

Description

Create a data frame with properties of all model variables, model units, and macros. Specify the variable types, variable names, and/or properties to get a subset of the data frame.

Usage

```
## S3 method for class 'sdbuildR_xmile'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  type = NULL,
  name = NULL,
  properties = NULL,
  ...
)
```

Arguments

`x`	A stock-and-flow model object of class `sdbuildR_xmile`.
`row.names`	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
`optional`	Ignored parameter.
`type`	Variable types to retain in the data frame. Must be one or more of 'stock', 'flow', 'constant', 'aux', 'gf', 'macro', or 'model_units'. Defaults to NULL to include all types.
`name`	Variable names to retain in the data frame. Defaults to NULL to include all variables.
`properties`	Variable properties to retain in the data frame. Defaults to NULL to include all properties.
`...`	Optional arguments

Value

A data.frame with one row per model component (variable, unit definition, or macro). Common columns include `type` (component type), `name` (variable name), `eqn` (equation), `units` (units of measurement), and `label` (descriptive label). Additional columns may include `to`, `from`, `non_negative`, and others depending on variable types. The exact columns returned depend on the `type` and `properties` arguments. Returns an empty data.frame if no components match the filters.

Examples

```
as.data.frame(xmile("SIR"))

# Only show stocks
as.data.frame(xmile("SIR"), type = "stock")

# Only show equation and label
as.data.frame(xmile("SIR"), properties = c("eqn", "label"))
```

`build` *Create, modify or remove variables*

Description

Add, change, or erase variables in a stock-and-flow model. Variables may be stocks, flows, constants, auxiliaries, or graphical functions.

Usage

```
build(
  sfm,
  name,
  type,
  eqn = "0.0",
  units = "1",
  label = name,
  doc = "",
  change_name = NULL,
  change_type = NULL,
  erase = FALSE,
  to = NULL,
  from = NULL,
  non_negative = FALSE,
  xpts = NULL,
  ypts = NULL,
  source = NULL,
  interpolation = "linear",
  extrapolation = "nearest",
  df = NULL
)
```

Arguments

`sfm`	Stock-and-flow model, object of class `sdbuildR_xmile`.
`name`	Variable name. Character vector.

type	Type of building block(s); one of 'stock', 'flow', 'constant', 'aux', or 'gf'). Does not need to be specified to modify an existing variable.
eqn	Equation (or initial value in the case of stocks). Defaults to "0.0".
units	Unit of variable, such as 'meter'. Defaults to "1" (no units).
label	Name of variable used for plotting. Defaults to the same as name.
doc	Description of variable. Defaults to "" (no description).
change_name	New name for variable (optional). Defaults to NULL to indicate no change.
change_type	New type for variable (optional). Defaults to NULL to indicate no change.
erase	If TRUE, remove variable from model. Defaults to FALSE.
to	Target of flow. Must be a stock in the model. Defaults to NULL to indicate no target.
from	Source of flow. Must be a stock in the model. Defaults to NULL to indicate no source.
non_negative	If TRUE, variable is enforced to be non-negative (i.e. strictly 0 or positive). Defaults to FALSE.
xpts	Only for graphical functions: vector of x-domain points. Must be of the same length as ypts.
ypts	Only for graphical functions: vector of y-domain points. Must be of the same length as xpts.
source	Only for graphical functions: name of the variable which will serve as the input to the graphical function. Necessary to specify if units are used. Defaults to NULL.
interpolation	Only for graphical functions: interpolation method. Must be either "constant" or "linear". Defaults to "linear".
extrapolation	Only for graphical functions: extrapolation method. Must be either "nearest" or "NA". Defaults to "nearest".
df	A data.frame with variable properties to add and/or modify. Each row represents one variable to build. Required columns depend on the variable type being created: • All types require: 'type', 'name' • Stocks require: 'eqn' (initial value) • Flows require: 'eqn', and at least one of 'from' or 'to' • Constants require: 'eqn' • Auxiliaries require: 'eqn' • Graphical functions require: 'xpts', 'ypts' Optional columns for all types: 'units', 'label', 'doc', 'non_negative' Optional columns for graphical functions: 'source', 'interpolation', 'extrapolation' Columns not applicable to a variable type should be set to NA. See Examples for a complete demonstration.

Value

A stock-and-flow model object of class sdbuildR_xmile

Stocks

Stocks define the state of the system. They accumulate material or information over time, such as people, products, or beliefs, which creates memory and inertia in the system. As such, stocks need not be tangible. Stocks are variables that can increase and decrease, and can be measured at a single moment in time. The value of a stock is increased or decreased by flows. A stock may have multiple inflows and multiple outflows. The net change in a stock is the sum of its inflows minus the sum of its outflows.

The obligatory properties of a stock are "name", "type", and "eqn". Optional additional properties are "units", "label", "doc", "non_negative".

Flows

Flows move material and information through the system. Stocks can only decrease or increase through flows. A flow must flow from and/or flow to a stock. If a flow is not flowing from a stock, the source of the flow is outside of the model boundary. Similarly, if a flow is not flowing to a stock, the destination of the flow is outside the model boundary. Flows are defined in units of material or information moved over time, such as birth rates, revenue, and sales.

The obligatory properties of a flow are "name", "type", "eqn", and either "from", "to", or both. Optional additional properties are "units", "label", "doc", "non_negative".

Constants

Constants are variables that do not change over the course of the simulation - they are time-independent. These may be numbers, but also functions. They can depend only on other constants.

The obligatory properties of a constant are "name", "type", and "eqn". Optional additional properties are "units", "label", "doc", "non_negative".

Auxiliaries

Auxiliaries are dynamic variables that change over time. They are used for intermediate calculations in the system, and can depend on other flows, auxiliaries, constants, and stocks.

The obligatory properties of an auxiliary are "name", "type", and "eqn". Optional additional properties are "units", "label", "doc", "non_negative".

Graphical functions

Graphical functions, also known as table or lookup functions, are interpolation functions used to define the desired output (y) for a specified input (x). They are defined by a set of x- and y-domain points, which are used to create a piecewise linear function. The interpolation method defines the behavior of the graphical function between x-points ("constant" to return the value of the previous x-point, "linear" to linearly interpolate between defined x-points), and the extrapolation method defines the behavior outside of the x-points ("NA" to return NA values outside of defined x-points, "nearest" to return the value of the closest x-point).

The obligatory properties of a graphical function are "name", "type", "xpts", and "ypts". "xpts" and "ypts" must be of the same length. Optional additional properties are "units", "label", "doc", "source", "interpolation", "extrapolation".

See Also

xmile()

Examples

```
# First initialize an empty model
sfm <- xmile()
summary(sfm)


# Add two stocks. Specify their initial values in the "eqn" property
# and their plotting label.
sfm <- build(sfm, "predator", "stock", eqn = 10, label = "Predator") |>
  build("prey", "stock", eqn = 50, label = "Prey")


# Add four flows: the births and deaths of both the predators and prey. The
# "eqn" property of flows represents the rate of the flow. In addition, we
# specify which stock the flow is coming from ("from") or flowing to ("to").
sfm <- build(sfm, "predator_births", "flow",
  eqn = "delta*prey*predator",
  label = "Predator Births", to = "predator"
) |>
  build("predator_deaths", "flow",
    eqn = "gamma*predator",
    label = "Predator Deaths", from = "predator"
  ) |>
  build("prey_births", "flow",
    eqn = "alpha*prey",
    label = "Prey Births", to = "prey"
  ) |>
  build("prey_deaths", "flow",
    eqn = "beta*prey*predator",
    label = "Prey Deaths", from = "prey"
  )
plot(sfm)

# The flows make use of four other variables: "delta", "gamma", "alpha", and
# "beta". Define these as constants in a vectorized manner for efficiency.
sfm <- build(sfm, c("delta", "gamma", "alpha", "beta"), "constant",
  eqn = c(.025, .5, .5, .05),
  label = c("Delta", "Gamma", "Alpha", "Beta"),
  doc = c(
    "Birth rate of predators", "Death rate of predators",
    "Birth rate of prey", "Death rate of prey by predators"
  )
)

# We now have a complete predator-prey model which is ready to be simulated.
sim <- simulate(sfm)
plot(sim)
```

```
# Modify a variable - note that we no longer need to specify type
sfm <- build(sfm, "delta", eqn = .03, label = "DELTA")

# Change variable name (throughout the model)
sfm <- build(sfm, "delta", change_name = "DELTA")

# Change variable type
sfm <- build(sfm, "DELTA", change_type = "stock")

# Remove variable
sfm <- build(sfm, "prey", erase = TRUE)

# To add and/or modify variables more quickly, pass a data.frame.
# The data.frame is processed row-wise.
# For instance, to create a logistic population growth model:
df <- data.frame(
  type = c("stock", "flow", "flow", "constant", "constant"),
  name = c("X", "inflow", "outflow", "r", "K"),
  eqn = c(.01, "r * X", "r * X^2 / K", 0.1, 1),
  label = c(
    "Population size", "Births", "Deaths", "Growth rate",
    "Carrying capacity"
  ),
  to = c(NA, "X", NA, NA, NA),
  from = c(NA, NA, "X", NA, NA)
)
sfm <- build(xmile(), df = df)

# Check for errors in the model
debugger(sfm)
```

`contains_IM` *Check if needle is in haystack*

Description

Check whether value is in vector or string. Equivalent of .Contains() in Insight Maker.

Usage

```
contains_IM(haystack, needle)
```

Arguments

`haystack`	Vector or string to search through
`needle`	Value to search for

Value

Logical value

Examples

```
contains_IM(c("a", "b", "c"), "d") # FALSE
contains_IM(c("abcdef"), "bc") # TRUE
```

`convert_u` *Convert unit in equation*

Description

In rare cases, it may be desirable to change the units of a variable within an equation. Use `convert_u()` to convert a variable to another matching unit. See `u()` for more information on the rules of specifying units. Note that units are only supported in Julia, not in R.

Usage

```
convert_u(x, unit_def)
```

Arguments

`x`	Variable
`unit_def`	Unit definition, e.g. u('seconds')

Value

Variable with new unit (only in Julia)

See Also

`model_units()`, `unit_prefixes()`, `u()`, `drop_u()`

Examples

```
# Change the unit of rate from minutes to hours
sfm <- xmile() |>
  build("rate", "constant", eqn = "10", units = "minutes") |>
  build("change", "flow",
    eqn = "(room_temperature - coffee_temperature) / convert_u(rate, u('hour'))"
  )
```

`debugger` *Debug stock-and-flow model*

Description

Check for common formulation problems in a stock-and-flow model.

Usage

```
debugger(sfm, quietly = FALSE)
```

Arguments

`sfm`	Stock-and-flow model, object of class `sdbuildR_xmile`.
`quietly`	If TRUE, don't print problems. Defaults to FALSE.

Details

The following problems are detected:

- An absence of stocks
- Flows without a source (`from`) or target (`to`)
- Flows connected to a stock that does not exist
- Undefined variable references in equations
- Circularity in equations
- Connected stocks and flows without both having units or no units
- Missing unit definitions

The following potential problems are detected:

- Absence of flows
- Stocks without inflows or outflows
- Equations with a value of 0

Value

If `quietly = FALSE`, list with problems and potential problems.

Examples

```
# No issues
sfm <- xmile("SIR")
debugger(sfm)

# Detect absence of stocks or flows
sfm <- xmile()
debugger(sfm)

# Detect stocks without inflows or outflows
sfm <- xmile() |> build("Prey", "stock")
debugger(sfm)

# Detect circularity in equation definitions
sfm <- xmile() |>
  build("Prey", "stock", eqn = "Predator") |>
  build("Predator", "stock", eqn = "Prey")
debugger(sfm)
```

---

drop_u *Drop unit in equation*

Description

In rare cases, it may be desirable to drop the units of a variable within an equation. Use drop_u() to render a variable unitless. See u() for more information on the rules of specifying units. Note that units are only supported in Julia, not in R.

Usage

```
drop_u(x)
```

Arguments

x	Variable with unit

Value

Unitless variable (only in Julia)

See Also

model_units(), unit_prefixes(), u(), convert_u()

Examples

```
# For example, the cosine function only accepts unitless arguments or
# arguments with units in radians or degrees
sfm <- xmile() |>
  build("a", "constant", eqn = "10", units = "minutes") |>
  build("b", "constant", eqn = "cos(drop_u(a))")
```

`ensemble` *Run ensemble simulations*

Description

Run an ensemble simulation of a stock-and-flow model, varying initial conditions and/or parameters in the range specified in `range`. The ensemble can be run in parallel using multiple threads by first setting `use_threads()`. The results are returned as a data.frame with summary statistics and optionally individual simulations.

Usage

```
ensemble(
  sfm,
  n = 10,
  return_sims = FALSE,
  range = NULL,
  cross = TRUE,
  quantiles = c(0.025, 0.975),
  only_stocks = TRUE,
  keep_nonnegative_flow = TRUE,
  keep_nonnegative_stock = FALSE,
  keep_unit = TRUE,
  verbose = TRUE
)
```

Arguments

`sfm`	Stock-and-flow model, object of class `sdbuildR_xmile`.
`n`	Number of simulations to run in the ensemble. When range is specified, n defines the number of simulations to run per condition. If each condition only needs to be run once, set n = 1. Defaults to 10.
`return_sims`	If TRUE, return the individual simulations in the ensemble. Set to FALSE to save memory. Defaults to FALSE.
`range`	A named list specifying parameter ranges for ensemble conditions. Names must correspond to existing stock or constant variable names in the model. Each list element should be a numeric vector of values to test. If cross = TRUE (default), all combinations of values are generated. For example, list(param1 = c(1, 2), param2 = c(10, 20)) creates 4 conditions: (1,10), (1,20), (2,10), (2,20).

If cross = FALSE, values are paired element-wise, requiring all vectors to have equal length. For example, list(param1 = c(1, 2, 3), param2 = c(10, 20, 30)) creates 3 conditions: (1,10), (2,20), (3,30). Defaults to NULL (no parameter variation).

`cross` If TRUE, cross the parameters in the range list to generate all possible combinations of parameters. Defaults to TRUE.

`quantiles` Quantiles to calculate in the summary, e.g. c(0.025, 0.975).

`only_stocks` If TRUE, only return stocks in output, discarding flows and auxiliaries. If FALSE, flows and auxiliaries are saved, which slows down the simulation. Defaults to FALSE.

`keep_nonnegative_flow`
If TRUE, keeps original non-negativity setting of flows. Defaults to TRUE.

`keep_nonnegative_stock`
If TRUE, keeps original non-negativity setting of stocks Defaults to FALSE.

`keep_unit` If TRUE, keeps units of variables. Defaults to TRUE.

`verbose` If TRUE, print details and duration of simulation. Defaults to TRUE.

Details

To run large simulations, it is recommended to limit the output size by saving fewer values. To create a reproducible ensemble simulation, set a seed using `sim_specs()`.

If you do not see any variation within a condition of the ensemble (i.e. the confidence bands are virtually non-existent), there are likely no random elements in your model. Without these, there can be no variability in the model. Try specifying a random initial condition or adding randomness to other model elements.

Value

Object of class `sdbuildR_ensemble`, which is a list containing:

success If TRUE, simulation was successful. If FALSE, simulation failed.

error_message If success is FALSE, contains the error message.

df data.frame with simulation results in long format, if return_sims is TRUE. The iteration number is indicated by column "i". If range was specified, the condition is indicated by column "j".

summary data.frame with summary statistics of the ensemble, including quantiles specified in quantiles. If range was specified, summary statistics are calculated for each condition (j) in the ensemble.

n Number of simulations run in the ensemble (per condition j if range is specified).

n_total Total number of simulations run in the ensemble (across all conditions if range is specified).

n_conditions Total number of conditions.

conditions data.frame with the conditions used in the ensemble, if range is specified.

init List with df (if return_sims = TRUE) and summary, containing data.frame with the initial values of the stocks used in the ensemble.

constants List with df (if return_sims = TRUE) and summary, containing data.frame with the constant parameters used in the ensemble.

script Julia script used for the ensemble simulation.

duration Duration of the simulation in seconds.

... Other parameters passed to ensemble

See Also

use_threads(), build(), xmile(), sim_specs(), use_julia()

Examples

```
# Load example and set simulation language to Julia
sfm <- xmile("predator_prey") |> sim_specs(language = "Julia")

# Set random initial conditions
sfm <- build(sfm, c("predator", "prey"), eqn = "runif(1, min = 20, max = 80)")

# For ensemble simulations, it is highly recommended to reduce the
# returned output. For example, to save only every 1 time units and discard
# the first 100 time units, use:
sfm <- sim_specs(sfm, save_at = 1, save_from = 100)

# Run ensemble simulation with 100 simulations
sims <- ensemble(sfm, n = 100)
plot(sims)

# Plot individual trajectories
sims <- ensemble(sfm, n = 10, return_sims = TRUE)
plot(sims, type = "sims")

# Specify which trajectories to plot
plot(sims, type = "sims", i = 1)

# Plot the median with lighter individual trajectories
plot(sims, central_tendency = "median", type = "sims", alpha = 0.1)

# Ensembles can also be run with exact values for the initial conditions
# and parameters. Below, we vary the initial values of the predator and the
# birth rate of the predators (delta). We generate a hunderd samples per
# condition. By default, the parameters are crossed, meaning that all
# combinations of the parameters are run.
sims <- ensemble(sfm,
  n = 50,
  range = list("predator" = c(10, 50), "delta" = c(.025, .05))
)

plot(sims)

# By default, a maximum of nine conditions is plotted.
# Plot specific conditions:
plot(sims, j = c(1, 3), nrows = 1)

# Generate a non-crossed design, where the length of each range needs to be
```

```
# equal:
sims <- ensemble(sfm,
  n = 10, cross = FALSE,
  range = list(
    "predator" = c(10, 20, 30),
    "delta" = c(.020, .025, .03)
  )
)
plot(sims, nrows = 3)

# Run simulation in parallel
use_threads(4)
sims <- ensemble(sfm, n = 10)

# Stop using threads
use_threads(stop = TRUE)

# Close Julia
use_julia(stop = TRUE)
```

`expit` *Expit function*

Description

Inverse of the logit function

Usage

```
expit(x)
```

Arguments

`x`	Numerical value

Value

Numerical value

Examples

```
expit(1)
```

`export_plot` *Save plot to a file*

Description

Save a plot of a stock-and-flow diagram or a simulation to a specified file path. Note that saving plots requires additional packages to be installed (see below).

Usage

```
export_plot(pl, file, width = 3, height = 4, units = "cm", dpi = 300)
```

Arguments

`pl`	Plot object.
`file`	File path to save plot to, including a file extension. For plotting a stock-and-flow model, the file extension can be one of png, pdf, svg, ps, eps, webp. For plotting a simulation, the file extension can be one of png, pdf, jpg, jpeg, webp. If no file extension is specified, it will default to png.
`width`	Width of image in units.
`height`	Height of image in units.
`units`	Units in which width and height are specified. Either "cm", "in", or "px".
`dpi`	Resolution of image. Only used if units is not "px".

Value

Returns `NULL` invisibly, called for side effects.

Examples

```
# Only if dependencies are installed
if (require("DiagrammeRsvg", quietly = TRUE) &
  require("rsvg", quietly = TRUE)) {
  sfm <- xmile("SIR")
  file <- tempfile(fileext = ".png")
  export_plot(plot(sfm), file)

  # Remove plot
  file.remove(file)
}


## Not run:
# requires internet
# Only if dependencies are installed
if (require("htmlwidgets", quietly = TRUE) &
  require("webshot2", quietly = TRUE)) {
```

```
  # Requires Chrome to save plotly plot:
  sim <- simulate(sfm)
  export_plot(plot(sim), file)

  # Remove plot
  file.remove(file)
}

## End(Not run)
```

`find_dependencies` *Find dependencies*

Description

Find which other variables each variable is dependent on.

Usage

```
find_dependencies(sfm, reverse = FALSE)
```

Arguments

`sfm`	Stock-and-flow model, object of class `sdbuildR_xmile`.
`reverse`	If FALSE, list for each variable X which variables Y it depends on for its equation definition. If TRUE, don't show dependencies but dependents. This reverses the dependencies, such that for each variable X, it lists what other variables Y depend on X.

Value

List, with for each model variable what other variables it depends on, or if `reverse = TRUE`, which variables depend on it

Examples

```
sfm <- xmile("SIR")
find_dependencies(sfm)
```

get_build_code *Generate code to build stock-and-flow model*

Description

Create R code to rebuild an existing stock-and-flow model. This may help to understand how a model is built, or to modify an existing one.

Usage

```
get_build_code(sfm)
```

Arguments

sfm	Stock-and-flow model, object of class sdbuildR_xmile.

Value

String with code to build stock-and-flow model from scratch.

Examples

```
sfm <- xmile("SIR")
get_build_code(sfm)
```

get_regex_time_units *Get regular expressions for time units in Julia*

Description

Get regular expressions for time units in Julia

Usage

```
get_regex_time_units()
```

Value

Named vector with regular expressions as names and units as entries

Examples

```
x <- get_regex_time_units()
head(x)
```

get_regex_units *Get regular expressions for units in Julia*

Description

Get regular expressions for units in Julia

Usage

```
get_regex_units(sfm = NULL)
```

Arguments

sfm	Stock-and-flow model, object of class sdbuildR_xmile.

Value

Named vector with regular expressions as names and units as entries

Examples

```
x <- get_regex_units()
head(x)
```

get_units *View all standard units*

Description

Obtain a data frame with all standard units in Julia's Unitful package and added custom units by sdbuildR.

Usage

```
get_units()
```

Value

A character matrix with 5 columns: description (unit description), name (unit symbol or abbreviation), full_name (full unit name), definition (mathematical definition in terms of base units), and prefix (logical indicating whether SI prefixes like kilo- or milli- can be applied). Includes SI base units, derived units, CGS units, US customary units, and custom units added by sdbuildR.

Examples

```
x <- get_units()
head(x)
```

header *Modify header of stock-and-flow model*

Description

The header of a stock-and-flow model contains metadata about the model, such as the name, author, and version. Modify the header of an existing model with standard or custom properties.

Usage

```
header(
  sfm,
  name = "My Model",
  caption = "My Model Description",
  created = Sys.time(),
  author = "Me",
  version = "1.0",
  URL = "",
  doi = "",
  ...
)
```

Arguments

sfm	Stock-and-flow model, object of class sdbuildR_xmile.
name	Model name. Defaults to "My Model".
caption	Model description. Defaults to "My Model Description".
created	Date the model was created. Defaults to Sys.time().
author	Creator of the model. Defaults to "Me".
version	Model version. Defaults to "1.0".
URL	URL associated with model. Defaults to "".
doi	DOI associated with the model. Defaults to "".
...	Optional other entries to add to the header.

Value

A stock-and-flow model object of class sdbuildR_xmile

Examples

```
sfm <- xmile() |>
  header(
    name = "My first model",
    caption = "This is my first model",
    author = "Kyra Evers",
    version = "1.1"
  )
```

indexof *Find index of needle in haystack*

Description

Find index of value in vector or string. Equivalent of .IndexOf() in Insight Maker.

Usage

```
indexof(haystack, needle)
```

Arguments

haystack	Vector or string to search through
needle	Value to search for

Value

Index, integer

Examples

```
indexof(c("a", "b", "c"), "b") # 2
indexof("haystack", "hay") # 1
indexof("haystack", "m") # 0
```

insightmaker_to_sfm *Import Insight Maker model*

Description

Import a stock-and-flow model from Insight Maker. Models may be your own or another user's. Importing causal loop diagrams or agent-based models is not supported.

Usage

```
insightmaker_to_sfm(
  URL,
  file,
  keep_nonnegative_flow = TRUE,
  keep_nonnegative_stock = FALSE,
  keep_solver = FALSE
)
```

Arguments

URL	URL to Insight Maker model. Character.
file	File path to Insight Maker model. Only used if URL is not specified. Needs to be a character with suffix .InsightMaker.
keep_nonnegative_flow	If TRUE, keeps original non-negativity setting of flows. Defaults to TRUE.
keep_nonnegative_stock	If TRUE, keeps original non-negativity setting of stocks Defaults to FALSE.
keep_solver	If TRUE, keep the ODE solver as it is. If FALSE, switch to Euler integration in case of non-negative stocks to reproduce the Insight Maker data exactly. Defaults to FALSE.

Details

Insight Maker models can be imported using either a URL or an Insight Maker file. Ensure the URL refers to a public (not private) model. To download a model file from Insight Maker, first clone the model if it is not your own. Then, go to "Share" (top right), "Export", and "Download Insight Maker file".

Value

A stock-and-flow model object of class sdbuildR_xmile

See Also

build(), xmile()

Examples

```
# Load a model from Insight Maker
sfm <- insightmaker_to_sfm(
  URL =
    "https://insightmaker.com/insight/43tz1nvUgbIiIOGSGtzIzj/Romeo-Juliet"
)
plot(sfm)



# Simulate the model
sim <- simulate(sfm)
plot(sim)
```

`install_julia_env` *Install, update, or remove Julia environment*

Description

Instantiate the Julia environment for sdbuildR to run stock-and-flow models using Julia. For more guidance, please see this vignette.

Usage

```
install_julia_env(remove = FALSE)
```

Arguments

`remove`	If TRUE, remove Julia environment for sdbuildR. This will delete the Manifest.toml file, as well as the SystemDynamicsBuildR.jl package. All other Julia packages remain untouched.

Details

`install_julia_env()` will:

- Start a Julia session
- Activate a Julia environment using sdbuildR's Project.toml
- Install SystemDynamicsBuildR.jl from GitHub (https://github.com/KCEvers/SystemDynamicsBuildR.jl)
- Install all other required Julia packages
- Create Manifest.toml
- Precompile packages for faster subsequent loading
- Stop the Julia session

Note that this may take 10-25 minutes the first time as Julia downloads and compiles packages.

Value

Invisibly returns NULL after instantiating the Julia environment.

See Also

`use_julia()`, `julia_status()`

Examples

```
## Not run:
  install_julia_env()

  # Remove Julia environment
  install_julia_env(remove = TRUE)

## End(Not run)
```

`julia_status` *Check status of Julia installation and environment*

Description

Check if Julia can be found and if the Julia environment for sdbuildR has been instantiated. Note that this does not mean a Julia session has been started, merely whether it *could* be. For more guidance, please see this vignette.

Usage

```
julia_status(verbose = TRUE)
```

Arguments

`verbose`	If TRUE, print detailed status information. Defaults to TRUE.

Value

A list with components:

`julia_found`	Logical. TRUE if Julia installation found.
`julia_version`	Character. Julia version string, or "" if not found.
`env_exists`	Logical. TRUE if Project.toml exists in sdbuildR package, which specifies the Julia packages and versions needed to instantiate the Julia environment for sdbuildR.
`env_instantiated`	Logical. TRUE if Manifest.toml exists (i.e., Julia environment was instantiated).
`status`	Character. Overall status: "julia_not_installed", "julia_needs_update", "sdbuildR_needs_reinstall", "install_julia_env", "ready", or "unknown".

What to Do Next

Based on the 'status' value:

"julia_not_installed" Install Julia from `https://julialang.org/install/`

"julia_needs_update" Update Julia to >= version 1.10

"install_julia_env" Run `install_julia_env()`

"ready" Run `use_julia()` to start a session

Examples

```
status <- julia_status()
print(status)
```

`length_IM` *Length of vector or string*

Description

Equivalent of .Length() in Insight Maker, which returns the number of elements when performed on a vector, but returns the number of characters when performed on a string

Usage

```
length_IM(x)
```

Arguments

`x`	A vector or a string

Value

The number of elements in x if x is a vector; the number of characters in x if x is a string

Examples

```
length_IM(c("a", "b", "c")) # 3
length_IM("abcdef") # 6
```

`logistic` *Logistic function*

Description

Computes the logistic (i.e., sigmoid) function with configurable slope, midpoint, and upper asymptote.

Usage

```
logistic(x, slope = 1, midpoint = 0, upper = 1)

sigmoid(x, slope = 1, midpoint = 0, upper = 1)
```

Arguments

x	Value at which to evaluate the function
slope	Slope of logistic function at the midpoint. Defaults to 1.
midpoint	Midpoint of logistic function where the output is upper/2. Defaults to 0.
upper	Upper asymptote (maximal value) of the logistic function. Defaults to 1.

Details

The logistic function is a smooth S-shaped curve bounded between 0 and upper. It transitions from near 0 to near upper around the midpoint, with the steepness of this transition controlled by slope.

Value

Numeric value given by

$$f(x) = \frac{upper}{1 + e^{-slope \cdot (x - midpoint)}}$$

Examples

```
logistic(0)
# equivalent:
sigmoid(0)
logistic(1, slope = 5, midpoint = 0.5, upper = 10)

# Visualize different slopes
x <- seq(-5, 5, length.out = 1000)
plot(x, logistic(x, slope = 1), type = "l", ylab = "f(x)", ylim = c(0, 1))
lines(x, logistic(x, slope = 5), col = "blue")
lines(x, logistic(x, slope = 50), col = "red")
legend("topleft", legend = c("slope = 1", "slope = 5", "slope = 50"),
       col = c("black", "blue", "red"), lty = 1)
```

---

logit *Logit function*

Description

Logit function

Usage

```
logit(p)
```

Arguments

p	Probability, numerical value between 0 and 1

Value

Numerical value

Examples

```
logit(.1)
```

`macro` *Create, modify or remove a global variable or function*

Description

Macros are global variables or functions that can be used throughout your stock-and-flow model. `macro()` adds, changes, or erases a macro.

Usage

```
macro(sfm, name, eqn = "0.0", doc = "", change_name = NULL, erase = FALSE)
```

Arguments

`sfm`	Stock-and-flow model, object of class `sdbuildR_xmile`.
`name`	Name of the macro. The equation will be assigned to this name.
`eqn`	Equation of the macro. A character vector. Defaults to "0.0".
`doc`	Documentation of the macro. Defaults to "".
`change_name`	New name for macro (optional). Defaults to NULL to indicate no change.
`erase`	If TRUE, remove macro from the model. Defaults to FALSE.

Value

A stock-and-flow model object of class `sdbuildR_xmile`

Examples

```
# Simple function
sfm <- xmile() |>
  macro("double", eqn = "function(x) x * 2") |>
  build("a", "constant", eqn = "double(2)")

# Function with defaults
sfm <- xmile() |>
  macro("scale", eqn = "function(x, factor = 10) x * factor") |>
  build("b", "constant", eqn = "scale(2)")

# If the logistic() function did not exist, you could create it yourself:
sfm <- macro(xmile(), "func", eqn = "function(x, slope = 1, midpoint = .5){
   1 / (1 + exp(-slope*(x-midpoint)))
```

```
}") |>
 build("c", "constant", eqn = "func(2, slope = 50)")
```

`model_units` *Create, modify or remove custom units*

Description

A large library of units already exists, but you may want to define your own custom units. Use `model_units()` to add, change, or erase custom units from a stock-and-flow model. Custom units may be new base units, or may be defined in terms of other (custom) units. See `u()` for more information on the rules of specifying units. Note that units are only supported in Julia, not in R.

Usage

```
model_units(sfm, name, eqn = "1", doc = "", erase = FALSE, change_name = NULL)
```

Arguments

`sfm`	Stock-and-flow model, object of class `sdbuildR_xmile`.
`name`	Name of unit. A character vector.
`eqn`	Definition of unit. String or vector of unit definitions. Defaults to "1" to indicate a base unit not defined in terms of other units.
`doc`	Documentation of unit.
`erase`	If TRUE, remove model unit from the model. Defaults to FALSE.
`change_name`	New name for model unit. Defaults to NULL to indicate no change.

Value

A stock-and-flow model object of class `sdbuildR_xmile`

See Also

`unit_prefixes()`

Examples

```
# Units are only supported with Julia
sfm <- xmile("Crielaard2022")
sfm <- model_units(sfm, "BMI", eqn = "kg/m^2", doc = "Body Mass Index")

# You may also use words rather than symbols for the unit definition.
# The following modifies the unit BMI:
sfm <- model_units(sfm, "BMI", eqn = "kilogram/meters^2")

# Remove unit:
```

```
sfm <- model_units(sfm, "BMI", erase = TRUE)

# Unit names may be changed to be syntactically valid and avoid overlap:
sfm <- model_units(xmile(), "C0^2")
```

`plot.sdbuildR_ensemble` *Plot timeseries of ensemble*

Description

Visualize ensemble simulation results of a stock-and-flow model. Either summary statistics or individual trajectories can be plotted. When multiple conditions j are specified, a grid of subplots is plotted. See `ensemble()` for examples.

Usage

```
## S3 method for class 'sdbuildR_ensemble'
plot(
  x,
  type = c("summary", "sims")[1],
  i = seq(1, min(c(x[["n"]], 10))),
  j = seq(1, min(c(x[["n_conditions"]], 9))),
  vars = NULL,
  add_constants = FALSE,
  nrows = ceiling(sqrt(max(j))),
  shareX = TRUE,
  shareY = TRUE,
  palette = "Dark 2",
  colors = NULL,
  font_family = "Times New Roman",
  font_size = 16,
  wrap_width = 25,
  showlegend = TRUE,
  j_labels = TRUE,
  central_tendency = c("mean", "median", FALSE)[1],
  central_tendency_width = 3,
  ...
)
```

Arguments

x	Output of `ensemble()`.
type	Type of plot. Either "summary" for a summary plot with mean or median lines and confidence intervals, or "sims" for individual simulation trajectories with mean or median lines. Defaults to "summary".

i	Indices of the individual trajectories to plot if type = "sims". Defaults to 1:10. Including a high number of trajectories will slow down plotting considerably.
j	Indices of the condition to plot. Defaults to 1:9. If only one condition is specified, the plot will not be a grid of subplots.
vars	Variables to plot. Defaults to NULL to plot all variables.
add_constants	If TRUE, include constants in plot. Defaults to FALSE.
nrows	Number of rows in the plot grid. Defaults to ceiling(sqrt(n_conditions)).
shareX	If TRUE, share the x-axis across subplots. Defaults to TRUE.
shareY	If TRUE, share the y-axis across subplots. Defaults to TRUE.
palette	Colour palette. Must be one of hcl.pals().
colors	Vector of colours. If NULL, the color palette will be used. If specified, will override palette. The number of colours must be equal to the number of variables in the simulation data frame. Defaults to NULL.
font_family	Font family. Defaults to "Times New Roman".
font_size	Font size. Defaults to 16.
wrap_width	Width of text wrapping for labels. Must be an integer. Defaults to 25.
showlegend	Whether to show legend. Must be TRUE or FALSE. Defaults to TRUE.
j_labels	Whether to plot labels indicating the condition of the subplot.
central_tendency	Central tendency to use for the mean line. Either "mean", "median", or FALSE to not plot the central tendency. Defaults to "mean".
central_tendency_width	Line width of central tendency. Defaults to 3.
...	Optional parameters

Value

Plotly object

See Also

ensemble()

`plot.sdbuildR_sim` *Plot timeseries of simulation*

Description

Visualize simulation results of a stock-and-flow model. Plot the evolution of stocks over time, with the option of also showing other model variables.

Usage

```
## S3 method for class 'sdbuildR_sim'
plot(
  x,
  add_constants = FALSE,
  vars = NULL,
  palette = "Dark 2",
  colors = NULL,
  font_family = "Times New Roman",
  font_size = 16,
  wrap_width = 25,
  showlegend = TRUE,
  ...
)
```

Arguments

x	Output of simulate().
add_constants	If TRUE, include constants in plot. Defaults to FALSE.
vars	Variables to plot. Defaults to NULL to plot all variables.
palette	Colour palette. Must be one of hcl.pals().
colors	Vector of colours. If NULL, the color palette will be used. If specified, will override palette. The number of colours must be equal to the number of variables in the simulation data frame. Defaults to NULL.
font_family	Font family. Defaults to "Times New Roman".
font_size	Font size. Defaults to 16.
wrap_width	Width of text wrapping for labels. Must be an integer. Defaults to 25.
showlegend	Whether to show legend. Must be TRUE or FALSE. Defaults to TRUE.
...	Optional parameters

Value

Plotly object

See Also

simulate(), as.data.frame.sdbuildR_sim(), plot.sdbuildR_xmile()

Examples

```
sfm <- xmile("SIR")
sim <- simulate(sfm)
plot(sim)

# The default plot title and axis labels can be changed like so:
plot(sim, main = "Simulated trajectory", xlab = "Time", ylab = "Value")
```

```
# Add constants to the plot
plot(sim, add_constants = TRUE)
```

`plot.sdbuildR_xmile` *Plot stock-and-flow diagram*

Description

Visualize a stock-and-flow diagram using the R package DiagrammeR. Stocks are represented as boxes. Flows are represented as arrows between stocks and/or double circles, where the latter represent what it outside of the model boundary. Thin grey edges indicate dependencies between variables. By default, constants (indicated by italic labels) are not shown. Hover over the variables to see their equations.

Usage

```
## S3 method for class 'sdbuildR_xmile'
plot(
  x,
  vars = NULL,
  format_label = TRUE,
  wrap_width = 20,
  font_size = 18,
  font_family = "Times New Roman",
  stock_col = "#83d3d4",
  flow_col = "#f48153",
  dependency_col = "#999999",
  show_dependencies = TRUE,
  show_constants = FALSE,
  show_aux = TRUE,
  minlen = 2,
  ...
)
```

Arguments

`x`	A stock-and-flow model object of class `sdbuildR_xmile`.
`vars`	Variables to plot. Defaults to NULL to plot all variables.
`format_label`	If TRUE, apply default formatting (removing periods and underscores) to labels if labels are the same as variable names.
`wrap_width`	Width of text wrapping for labels. Must be an integer. Defaults to 20.
`font_size`	Font size. Defaults to 18.
`font_family`	Font name. Defaults to "Times New Roman".
`stock_col`	Colour of stocks. Defaults to "#83d3d4".

flow_col	Colour of flows. Defaults to "#f48153".
dependency_col	Colour of dependency arrows. Defaults to "#999999".
show_dependencies	If TRUE, show dependencies between variables. Defaults to TRUE.
show_constants	If TRUE, show constants. Defaults to FALSE.
show_aux	If TRUE, show auxiliary variables. Defaults to TRUE.
minlen	Minimum length of edges; must be an integer. Defaults to 2.
...	Optional arguments

Value

Stock-and-flow diagram

See Also

insightmaker_to_sfm(), xmile(), plot.sdbuildR_sim()

Examples

```
sfm <- xmile("SIR")
plot(sfm)

# Don't show constants or auxiliaries
plot(sfm, show_constants = FALSE, show_aux = FALSE)

# Only show specific variables
plot(sfm, vars = "Susceptible")
```

---

print.summary.sdbuildR_xmile

Print method for summary.sdbuildR_xmile

Description

Print method for summary.sdbuildR_xmile

Usage

```
## S3 method for class 'summary.sdbuildR_xmile'
print(x, ...)
```

Arguments

x	A summary object of class summary.sdbuildR_xmile
...	Additional arguments (unused)

Value

Invisibly returns the summary object of class summary.sdbuildR_xmile

---

`pulse` *Create pulse function*

Description

Create a pulse function that jumps from zero to a specified height at a specified time, and returns to zero after a specified width. The pulse can be repeated at regular intervals.

Usage

```
pulse(times, start, height = 1, width = 1, repeat_interval = NULL)
```

Arguments

`times`	Vector of simulation times
`start`	Start time of pulse in simulation time units.
`height`	Height of pulse. Defaults to 1.
`width`	Width of pulse in simulation time units. This cannot be equal to or less than 0. To indicate an instantaneous pulse, specify the simulation step size.
`repeat_interval`	Interval at which to repeat pulse. Defaults to NULL to indicate no repetition.

Details

Equivalent of Pulse() in Insight Maker

Value

Pulse interpolation function

See Also

`step()`, `ramp()`, `seasonal()`

Examples

```
# Create a simple model with a pulse function
# that starts at time 5, jumps to a height of 2
# with a width of 1, and does not repeat
sfm <- xmile() |>
  build("a", "stock") |>
  # Specify the global variable "times" as simulation times
  build("input", "constant", eqn = "pulse(times, 5, 2, 1)") |>
  build("inflow", "flow", eqn = "input(t)", to = "a")
```

```
sim <- simulate(sfm, only_stocks = FALSE)
plot(sim)

# Create a pulse that repeats every 5 time units
sfm <- build(sfm, "input", eqn = "pulse(times, 5, 2, 1, 5)")

sim <- simulate(sfm, only_stocks = FALSE)
plot(sim)
```

`ramp` *Create ramp function*

Description

Create a ramp function that increases linearly from 0 to a specified height at a specified start time, and stays at this height after the specified end time.

Usage

```
ramp(times, start, finish, height = 1)
```

Arguments

`times`	Vector of simulation times
`start`	Start time of ramp
`finish`	End time of ramp
`height`	End height of ramp, defaults to 1

Details

Equivalent of Ramp() in Insight Maker

Value

Ramp interpolation function

See Also

`step()`, `pulse()`, `seasonal()`

Examples

```
# Create a simple model with a ramp function
sfm <- xmile() |>
  build("a", "stock") |>
  # Specify the global variable "times" as simulation times
  build("input", "constant", eqn = "ramp(times, 20, 30, 3)") |>
  build("inflow", "flow", eqn = "input(t)", to = "a")



sim <- simulate(sfm, only_stocks = FALSE)
plot(sim)

# To create a decreasing ramp, set the height to a negative value
sfm <- build(sfm, "input", eqn = "ramp(times, 20, 30, -3)")

sim <- simulate(sfm, only_stocks = FALSE)
plot(sim)
```

---

rbool *Generate random logical value*

Description

Equivalent of RandBoolean() in Insight Maker

Usage

```
rbool(p)
```

Arguments

p	Probability of TRUE, numerical value between 0 and 1

Value

Logical value

Examples

```
rbool(.5)
```

rdist *Generate random number from custom distribution*

Description

Equivalent of RandDist() in Insight Maker

Usage

```
rdist(a, b)
```

Arguments

a	Vector to draw sample from
b	Vector of probabilities

Value

One sample from custom distribution

Examples

```
rdist(c(1, 2, 3), c(.5, .25, .25))
```

rem *Remainder and modulus*

Description

Remainder and modulus operators. The modulus and remainder are not the same in case either a or b is negative. If you work with negative numbers, modulus is always non-negative (it matches the sign of the divisor).

Usage

```
rem(a, b)

mod(a, b)

a %REM% b
```

Arguments

a	Dividend
b	Divisor

Value

Remainder

Examples

```
# Modulus and remainder are the same when a and b are positive
a <- 7
b <- 3
rem(a, b)
mod(a, b)
# Modulus and remainder are NOT when either a or b is negative
a <- -7
b <- 3
rem(a, b)
mod(a, b)
a <- 7
b <- -3
rem(a, b)
mod(a, b)
# Modulus and remainder are the same when both a and b are negative
a <- -7
b <- -3
rem(a, b)
mod(a, b)

# Alternative way of computing the remainder:
a %REM% b
```

---

round_IM *Round Half-Up (as in Insight Maker)*

Description

R rounds .5 to 0, whereas Insight Maker rounds .5 to 1. This function is the equivalent of Insight Maker's Round() function.

Usage

```
round_IM(x, digits = 0)
```

Arguments

x	Value
digits	Number of digits; optional, defaults to 0

Value

Rounded value

Examples

```
round_IM(.5) # 1
round(.5) # 0
round_IM(-0.5) # 0
round(-0.5) # 0
round_IM(1.5) # 2
round(1.5) # 2
```

`seasonal` *Create a seasonal wave function*

Description

Create a seasonal wave function that oscillates between -1 and 1, with a specified period and shift. The wave peaks at the specified shift time.

Usage

```
seasonal(times, period = 1, shift = 0)
```

Arguments

`times`	Vector of simulation times
`period`	Duration of wave in simulation time units. Defaults to 1.
`shift`	Timing of wave peak in simulation time units. Defaults to 0.

Details

Equivalent of Seasonal() in Insight Maker

Value

Seasonal interpolation function

See Also

`step()`, `pulse()`, `ramp()`

Examples

```
# Create a simple model with a seasonal wave
sfm <- xmile() |>
  build("a", "stock") |>
  # Specify the global variable "times" as simulation times
  build("input", "constant", eqn = "seasonal(times, 10, 0)") |>
  build("inflow", "flow", eqn = "input(t)", to = "a")

sim <- simulate(sfm, only_stocks = FALSE)
plot(sim)
```

`simulate` *Simulate stock-and-flow model*

Description

Simulate a stock-and-flow model with simulation specifications defined by `sim_specs()`. If `sim_specs(language = "julia")`, the Julia environment will first be set up with `use_julia()`. If any problems are detected by `debugger()`, the model cannot be simulated.

Usage

```
simulate(
  sfm,
  keep_nonnegative_flow = TRUE,
  keep_nonnegative_stock = FALSE,
  keep_unit = TRUE,
  only_stocks = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`sfm`	Stock-and-flow model, object of class `sdbuildR_xmile`.
`keep_nonnegative_flow`	If TRUE, keeps original non-negativity setting of flows. Defaults to TRUE.
`keep_nonnegative_stock`	If TRUE, keeps original non-negativity setting of stocks Defaults to FALSE.
`keep_unit`	If TRUE, keeps units of variables. Defaults to TRUE.
`only_stocks`	If TRUE, only return stocks in output, discarding flows and auxiliaries. If FALSE, flows and auxiliaries are saved, which slows down the simulation. Defaults to FALSE.
`verbose`	If TRUE, print duration of simulation. Defaults to FALSE.
`...`	Optional arguments

Value

Object of class `sdbuildR_sim`, a list containing:

df Data frame: simulation results (time, variable, value)

init Named vector: initial stock values

constants Named vector: constant parameters

script Character: generated simulation code (R or Julia)

duration Numeric: simulation time in seconds

success Logical: TRUE if completed without errors

... Other parameters passed to simulate

Use `as.data.frame()` to extract results, `plot()` to visualize.

See Also

`build()`, `xmile()`, `debugger()`, `sim_specs()`, `use_julia()`

Examples

```
sfm <- xmile("SIR")
sim <- simulate(sfm)
plot(sim)

# Obtain all model variables
sim <- simulate(sfm, only_stocks = FALSE)
plot(sim, add_constants = TRUE)


# Use Julia for models with units
sfm <- sim_specs(xmile("coffee_cup"), language = "Julia")
sim <- simulate(sfm)
plot(sim)

# Close Julia session
use_julia(stop = TRUE)
```

---

sim_specs *Modify simulation specifications*

Description

Simulation specifications are the settings that determine how the model is simulated, such as the integration method (i.e. solver), start and stop time, and timestep. Modify these specifications for an existing stock-and-flow model.

Usage

```
sim_specs(
  sfm,
  method = "euler",
  start = "0.0",
  stop = "100.0",
  dt = "0.01",
  save_at = dt,
  save_from = start,
  seed = NULL,
```

```
  time_units = "s",
  language = "R"
)
```

Arguments

sfm	Stock-and-flow model, object of class sdbuildR_xmile.
method	Integration method. Defaults to "euler".
start	Start time of simulation. Defaults to 0.
stop	End time of simulation. Defaults to 100.
dt	Timestep of solver; controls simulation accuracy. Smaller = more accurate but slower. Defaults to 0.01.
save_at	Timestep at which to save computed values; controls output size. Must be >= dt. Use larger than dt to reduce memory without sacrificing accuracy. Example: dt = 0.01, save_at = 1 gives accurate simulation but only saves every 100th point. Defaults to dt (save everything).
save_from	Time at which to start saving values. Use to discard initial transient behavior. Must be >= start. Defaults to start.
seed	Seed number to ensure reproducibility across runs in case of random elements. Must be an integer. Defaults to NULL (no seed).
time_units	Simulation time unit, e.g. 's' (second). Defaults to "s".
language	Coding language in which to simulate model. Either "R" or "Julia". Julia is necessary for using units or delay functions. Defaults to "R".

Value

A stock-and-flow model object of class sdbuildR_xmile

See Also

solvers()

Examples

```
sfm <- xmile("predator_prey") |>
  sim_specs(start = 0, stop = 50, dt = 0.1)
sim <- simulate(sfm)
plot(sim)

# Change the simulation method to "rk4"
sfm <- sim_specs(sfm, method = "rk4")

# Change the time units to "years", such that one time unit is one year
sfm <- sim_specs(sfm, time_units = "years")

# To save storage but not affect accuracy, use save_at and save_from
sfm <- sim_specs(sfm, save_at = 1, save_from = 10)
sim <- simulate(sfm)
```

```
head(as.data.frame(sim))

# Add stochastic initial condition but specify seed to obtain same result
sfm <- sim_specs(sfm, seed = 1) |>
  build(c("predator", "prey"), eqn = "runif(1, 20, 50)")

# Change the simulation language to Julia to use units
sfm <- sim_specs(sfm, language = "Julia")
```

`solvers` *Check or translate between deSolve and Julia DifferentialEquations solvers*

Description

This function either checks whether a solver method exists or provides bidirectional translation between R's deSolve package solvers and Julia's DifferentialEquations.jl solvers.

Usage

```
solvers(method, from = c("R", "Julia"), to = NULL, show_info = FALSE)
```

Arguments

`method`	Character string of solver name
`from`	Character string indicating source language: "R" or "Julia"
`to`	Character string indicating target language: "R" or "Julia"
`show_info`	Logical, whether to display additional solver information

Value

Character vector of equivalent solver(s) or list with details

Examples

```
# Translate from R to Julia
solvers("euler", from = "R", to = "Julia")
solvers("rk45dp6", from = "R", to = "Julia")

# Translate from Julia to R
solvers("Tsit5", from = "Julia", to = "R")
solvers("DP5", from = "Julia", to = "R", show_info = TRUE)

# List all available solvers
solvers(from = "R")
solvers(from = "Julia")
```

step *Create step function*

Description

Create a step function that jumps from zero to a specified height at a specified time, and remains at that height until the end of the simulation time.

Usage

```
step(times, start, height = 1)
```

Arguments

times	Vector of simulation times
start	Start time of step
height	Height of step, defaults to 1

Details

Equivalent of Step() in Insight Maker

Value

Step interpolation function

See Also

ramp(), pulse(), seasonal()

Examples

```
# Create a simple model with a step function
# that jumps at time 50 to a height of 5
sfm <- xmile() |>
  build("a", "stock") |>
  # Specify the global variable "times" as simulation times
  build("input", "constant", eqn = "step(times, 50, 5)") |>
  build("inflow", "flow", eqn = "input(t)", to = "a")



sim <- simulate(sfm, only_stocks = FALSE)
plot(sim)

# Negative heights are also possible
sfm <- build(sfm, "input", eqn = "step(times, 50, -10)")

sim <- simulate(sfm, only_stocks = FALSE)
plot(sim)
```

summary.sdbuildR_xmile *Print overview of stock-and-flow model*

Description

Print summary of stock-and-flow model, including number of stocks, flows, constants, auxiliaries, graphical functions, macros, and custom model units, as well as simulation specifications and use of delay functions.

Usage

```
## S3 method for class 'sdbuildR_xmile'
summary(object, ...)
```

Arguments

object	A stock-and-flow model object of class sdbuildR_xmile
...	Optional arguments

Value

Summary object of class summary.sdbuildR_xmile

See Also

build()

Examples

```
sfm <- xmile("SIR")
summary(sfm)
```

u *Specify unit in equations*

Description

Flexibly use units in equations by enclosing them in u(). Note that units are only supported in Julia, not in R.

Usage

```
u(unit_str)
```

Arguments

unit_str	Unit string; e.g. '3 seconds'

Details

Unit strings are converted to their standard symbols using regular expressions. This means that you can easily specify units without knowing their standard symbols. For example, u('kilograms per meters squared') will become 'kg/m^2'. You can use title-case for unit names, but letters cannot all be uppercase if this is not the standard symbol. For example, 'kilogram' works, but 'KILOGRAM' does not. This is to ensure that the right unit is detected.

Value

Specified unit (only in Julia)

See Also

model_units(), unit_prefixes(), convert_u(), drop_u()

Examples

```
# Use units in equations
sfm <- xmile() |>
  build("a", "constant",
    eqn = "u('10kilometers') - u('3meters')",
    units = "centimeters"
  )

# Units can also be set by multiplying a number with a unit
sfm <- xmile() |>
  build("a", "constant", eqn = "10 * u('kilometers') - u('3meters')")

# Addition and subtraction is only allowed between matching units
sfm <- xmile() |>
  build("a", "constant", eqn = "u('3seconds') + u('1hour')")

# Division, multiplication, and exponentiation are allowed between different units
sfm <- xmile() |>
  build("a", "constant", eqn = "u('10grams') / u('1minute')")

# Use custom units in equations
sfm <- xmile() |>
  model_units("BMI", eqn = "kilograms/meters^2", doc = "Body Mass Index") |>
  build("weight_gain", "flow", eqn = "u('2 BMI / year')", units = "BMI/year")

# Unit strings are often needed in flows to ensure dimensional consistency
sfm <- xmile() |>
  sim_specs(stop = 1, time_units = "days") |>
  build("consumed_food", "stock", eqn = "1", units = "kilocalories") |>
  build("eating", "flow",
    eqn = "u('750kilocalories') / u('6hours')",
```

```
  units = "kilocalories/day", to = "consumed_food"
)
```

`unit_prefixes` *Show unit prefixes*

Description

Show unit prefixes

Usage

```
unit_prefixes()
```

Value

A character matrix with 3 columns: `prefix` (prefix name like "kilo" or "micro"), `symbol` (prefix symbol like "k"), and `scale` (power-of-ten multiplier like "10^3" or "10^-6"). Rows are ordered from largest (yotta, 10^24) to smallest (yocto, 10^-24).

Examples

```
unit_prefixes()
```

`url_to_IM` *Extract Insight Maker model from URL*

Description

Create XML string from Insight Maker URL. For internal use; use `insightmaker_to_sfm()` to import an Insight Maker model.

Usage

```
url_to_IM(URL, file = NULL)
```

Arguments

`URL`	String with URL to an Insight Maker model
`file`	If specified, file path to save Insight Maker model to. If NULL, do not save model.

Value

XML string with Insight Maker model

See Also

insightmaker_to_sfm()

Examples

```
xml <- url_to_IM(
  URL =
    "https://insightmaker.com/insight/43tz1nvUgbIiIOGSGtzIzj/Romeo-Juliet"
)
```

---

use_julia *Start Julia and activate environment*

Description

Start Julia session and activate Julia environment to simulate stock-and-flow models. To do so, Julia needs to be installed and findable from within R. See this vignette for guidance. In addition, the Julia environment specifically for sdbuildR needs to have been instantiated. This can be set up with `install_julia_env()`.

Usage

```
use_julia(stop = FALSE, force = FALSE)
```

Arguments

`stop`	If TRUE, stop active Julia session. Defaults to FALSE.
`force`	If TRUE, force Julia setup to execute again.

Details

Julia supports running stock-and-flow models with units as well as ensemble simulations (see `ensemble()`).

In every R session, `use_julia()` needs to be run once (which is done automatically in `simulate()`), which can take around 30-60 seconds.

Value

Returns `NULL` invisibly, used for side effects

See Also

julia_status(), install_julia_env()

Examples

```
# Start a Julia session and activate the Julia environment for sdbuildR
use_julia()

# Stop Julia session
use_julia(stop = TRUE)
```

---

use_threads *Set up threaded ensemble simulations*

Description

Specify the number of threads for ensemble simulations in Julia. This will not overwrite your current global setting for JULIA_NUM_THREADS. Note that this does not affect regular simulations with simulate().

Usage

```
use_threads(n = parallel::detectCores() - 1, stop = FALSE)
```

Arguments

n	Number of Julia threads to use. Defaults to parallel::detectCores() - 1. If set to a value higher than the number of available cores minus 1, it will be set to the number of available cores minus 1.
stop	Stop using threaded ensemble simulations. Defaults to FALSE.

Value

No return value, called for side effects

See Also

ensemble(), use_julia()

Examples

```
# Use Julia with 4 threads
use_julia()
use_threads(n = 4)

# Stop using threads
use_threads(stop = TRUE)

# Stop using Julia
use_julia(stop = TRUE)
```

`xmile` *Create a new stock-and-flow model*

Description

Initialize a stock-and-flow model of class `sdbuildR_xmile`. You can either create an empty stock-and-flow model or load a template from the model library.

Usage

```
xmile(name = NULL)
```

Arguments

`name`	Name of the template to load. If `NULL`, an empty stock-and-flow model will be created with default simulation parameters and a default header. If specified, `name` should be one of the available templates: • **logistic_model**: Population growth with carrying capacity • **SIR**: Epidemic model (Susceptible-Infected-Recovered) • **predator_prey**: Lotka-Volterra dynamics • **cusp**: Cusp catastrophe model • **Crielaard2022**: Eating behavior (doi: 10.1037/met0000484) • **coffee_cup**: Temperature equilibration (Meadows) • **bank_account**: Compound interest (Meadows) • **Lorenz**: Lorenz attractor (chaotic) • **Rossler**: Rossler attractor (chaotic) • **vanderPol**: Van der Pol oscillator • **Duffing**: Forced Duffing oscillator • **Chua**: Chua's circuit (chaotic) • **JDR**: Job Demands-Resources Theory as formalized in Evers et al. (submitted)

Details

Do not edit the object manually; this will likely lead to errors downstream. Rather, use `header()`, `sim_specs()`, `build()`, `macro()`, and `model_units()` for safe manipulation.

Value

A stock-and-flow model object of class `sdbuildR_xmile`. Its structure is based on XML Interchange Language for System Dynamics (XMILE). It is a nested list, containing:

header Meta-information about model. A list containing arguments listed in `header()`.

sim_specs Simulation specifications. A list containing arguments listed in `sim_specs()`.

model Model variables, grouped under the variable types stock, flow, aux (auxiliaries), constant, and gf (graphical functions). Each variable contains arguments as listed in `build()`.

macro Global variable or functions. A list containing arguments listed in `macro()`.

model_units Custom model units. A list containing arguments listed in `model_units()`.

Use `summary()` to summarize, `as.data.frame()` to convert to a data.frame, `plot()` to visualize.

See Also

`build()`, `header()`, `macro()`, `model_units()`, `sim_specs()`

Examples

```
sfm <- xmile()
summary(sfm)



# Load a template
sfm <- xmile("Lorenz")
sim <- simulate(sfm)
plot(sim)
```

Index