sdbuildR: Building System Dynamics Models in R

true true

2022-11-09

Abstract

Complex problems like climate change, geopolitical conflict, and disease have no straight-forward solutions. The field of System Dynamics aims to understand and intervene on complex systems by taking a feedback-centered view, where the interactions, nonlinearities, and delays within a system are central to addressing its problems. These systems can be mathematically described with differential equations as stock-and-flow models. Software to build and simulate stock-and-flow models is predominantly not open-access or written in domain-specific language, limiting the accessibility of this powerful technique. This paper introduces the package *sdbuildR* to build and simulate stock-and-flow models in an accessible and flexible manner in R. System Dynamics modelling in R offers five main benefits: accessibility, compatibility, flexibility, scalability, and reproducibility. In addition, *sdbuildR* can translate stock-and-flow models created in Insight Maker, an open-access online System Dynamics tool. By promoting System Dynamics modelling in R, sdbuildR aims to support a collaborative, open-source approach to further integrate and advance System Dynamics tools.

Introduction

Today's complex problems - the climate, disease, social divide - arise within interconnected, dynamic systems. A simple, linear understanding of the problem often does not suffice, as the relation between cause and effect is rarely proportional or immediate. Complex problems require a systems approach, which considers behaviour to arise from interactions between system components over time (**kanti2017?**). Offering tools to understand and model this complexity, System Dynamics takes a feedback-centered view, where the interactions, nonlinearities, and delays within a system are central to addressing its problems. By building and simulating stock-and-flow models consisting of a set of differential equations, System Dynamics seeks to create more effective interventions. Used in fields like ecology, economics, epidemiology, and social sciences, System Dynamics has been used to design interventions for the NHS (**LaneHusemann2008?**), polioviruses (**Thompson2008?**), and the Dutch energy transition (**Gooyert2016?**), among others.

Building and testing stock-and-flow models requires extensive software. Historically, System Dynamics software has been licensed and proprietary, such as Vensim, Stella, and Powersim (**stella?**). Despite the rise of open-access System Dynamics software in the past decade, in practice, accessibility remains limited when software is not easily incorporated into the platforms most researchers are used to and skilled in. Moreover, a domain-specific language limits the integration of new methodologies, such as those offered by machine learning and artificial intelligence (**XXX?**).

R is well-suited to enable accessible, open-source System Dynamics modelling. In R, packages are easily distributed and integrated within one's workflow, facilitating collaborative research which makes use of recent software developments. The flexibility in data wrangling and visualization in R can further enhance the ease of use and communication of System Dynamic model results. R also supports computationally intensive analyses using parallel computing, allowing for extensive sensitivity testing and scenario simulation of System Dynamics models. Moreover, R could further enhance the robustness of System Dynamics models, as it is a leading platform for reproducible research {(**XXX?**)}, with robust version control and easily shareable scripts.

The accessibility, compatibility, flexibility, reproducibility, and scaleability offered by R could thus advance System Dynamics modelling. Likewise, the objective of System Dynamics to design better interventions could be particularly appealing to the R community, spanning fields such as social science, bioinformatics, statistics, and data science (**schilder?**). Nevertheless, System Dynamics has seen very limited engagement within the R community. Few resources for System Dynamics modelling in R exist, two notable exceptions being the book *System Dynamics Modelling with R* by Jim Duggan and the R package *readsdr* (https://github.com/jandraor/readsdr). Duggan provides an incredibly useful introduction to System Dynamics models, yet a book may be too daunting for new users to engage with System Dynamics. The package *readsdr* translates System Dynamics models from licensed software to R, and includes sensitivity and policy testing. However, the translation is limited to simple models, and the model is not easily modified without knowledge of stock-and-flow models. For more complicated models and projects requiring model flexibility, this leaves a dependency on licensed software, which aids the user to a much greater degree.

To help connect the field of System Dynamics with the R community, this paper introduces *sdbuildR*, a package to build and simulate stock-and-flow models in R. The package is designed to lower the barrier to System Dynamics modelling, emphasizing ease of use with little prior knowledge required. Model components can easily be modified, as the simulation script is compiled anew. *sdbuildR* also supports the translation of models created in Insight Maker, an open-access online tool to create, simulate, and analyse System Dynamics models. To ensure correct implementation, *sdbuildR* has been validated with over a hundred Insight Maker models. After a brief introduction into the basics of System Dynamics modelling, we illustrate how to use *sdbuildR* and its capabilities, with the aim of promoting and facilitating System Dynamics modelling in R.

Basics of System Dynamics

Introduced in the 1950s by MIT professor Jay Forrester, System Dynamics is both a conceptual tool and a modelling technique to understand the structure and dynamics of complex systems (**sterman2000?**). A central axiom of System Dynamics modelling is that the structure of a system generates its behaviour: A system's components, interactions, nonlinearities, and delays are responsible for the events that occur over time. System Dynamics focuses on the *feedback loops* that permeate this structure. A feedback loop is "a closed chain of causal connections" (**Meadows?**) through which a variable affects itself. A simple example is how the amount of your savings determines how much they increase through interest (**Meadows?**). Feedback structures also involve information feedback, where for example drug-related crime can increase through a positive feedback loop including more police action, more arrests, less supply, higher prices, and more drug-related crime (**morecroft2015?**). Feedback thinking is directly relevant to intervention design: A major insight of System Dynamics is that hidden feedback structures commonly underlie policy resistance, unintended consequences, and counterintuitive behaviour (**sterman2000?**). By mapping out feedback structures, System Dynamics modelling is a tool to anticipate delayed and undesirable responses to interventions and design more effective policies (**sterman2000?**).

The feedback-centred view illustrates that System Dynamics seeks an *endogenous* explanation for a problem, which locates XX arises from the interactions within the system itself. An *exogenous* variable acts on the system but is not affected by the system in turn. In the drug-related crime example, treating police action as an exogenous factor rather than endogenous would mean that police action does not increase if drug-related crime increases; causal influence would be unidirectional. Aside from endogenous and exogenous variables, many variables in a System Dynamics model are *excluded.* As any model is a simplification of reality, it will necessarily exclude variables that doubtlessly influence the system, but are less relevant for the problem at hand. A System Dynamics model thus communicates a *model boundary* constraining the variables that are being modelled.

The hypothesized causal structure of a system is commonly visualised in *Causal Loop Diagrams* (CLD). As a running example throughout the rest of the paper, we will use a modified version of Crielaard et al.'s (2022) model of compensatory behaviour in eating disorders (**crielaard2022?**). In disordered eating, compensatory behaviours are acts to prevent weight gain or relieve feelings of guilt after eating, and include self-induced vomiting, laxative use and overexercise (**Stiles-Shields2012?**) Crielaard et al.'s (2022) CLD maps out the feedback loops of Food intake, Hunger, and Compensatory behaviour, with the addition of Eating triggers.[1] A feedback loop can be balancing (i.e. counteracting, goal-seeking, negative) or reinforcing (i.e. amplifying, destabilizing, positive). In this CLD, there are four negative feedback loops: between Food intake and Hunger, between Hunger and Compensatory behaviour, between Food intake and Compensatory behaviour, and Food intake with itself. For details explaining the relation between these variables, please see Crielaard et al. (2022).

Stock-and-flow models

A CLD conveys the feedback structures governing the behaviour of the system. However, a CLD merely describes the structure of the system on a conceptual level. In order to simulate the behaviour of the system over time, a mathematical model is needed. A *stock-and-flow* model is one way to formally describe a system with a set of differential equations which model how the system changes over time. As a first step, the units of each variable need to be defined. Units, such as seconds, kilocalories, or people over years, quantify each variable, and help to interpret and formulate interactions between variables. *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 (**sterman?**). Stocks are variables that can increase and decrease, and be measured at a single moment in time. The value of a Stock is increased or decreased by *Flows*. Flows move material through the system. An inflow increases a Stock, and an outflow decreases a Stock, such that the net change in a Stock is the sum of its inflows minus the sum of its outflows. Flows are defined in units of material moved over time, such as birth rates, revenue, and sales. Material may flow from and to other Stocks, or may originate from or disappear to an unspecified *source* or *sink* outside of the model. Variables included in the model are endogenous, whereas sources and sinks which are outside of the model boundary are exogenous. An exogenous source or sink is in actuality also a Stock, but is not included in the model for simplicity and parsimony (**meadows?**). As the dynamics of sinks and sources are not modelled, their capacity is infinite (**sterman?**). In addition to Stocks and Flows, a System Dynamics model may also contain two other building blocks: any other static parameters or intermediately computed variables are *Auxiliaries*, and any user-defined linear interpolation functions are *Graphical Functions* (also called converters, or table or lookup functions).

Table 1: System Dynamics Building Blocks

Building Block	Purpose	Example
Stock	Define the state of the system	People, products, beliefs, blood sugar, CO2 in atmosphere
Flow	Move material between Stocks and outside the model boundary	Birth rates, product order rates, revenue, sales
Auxiliary	Define static parameters or dynamic intermediate computation	Fixed prices, sum of all Stocks, ratios of flows
Graphical Function	Custom interpolation functions	Interventions or events occurring at known times such as vaccines or storms

Figure **??** shows the CLD translated to a stock-and-flow model. In Crielaard et al.'s (2022) model, the Stocks are Food intake, Hunger, and Compensatory behaviour. Note that a direct feedback loop between two variables in a CLD does not necessarily translate to Flows directly connecting two Stocks. If no material directly flows from one Stock to another, two Stocks are not directly connected by a Flow. Rather, the type of feedback is *information* feedback, where a Stock directly affects the inflow or outflow of another Stock. For example, in the feedback loop between Hunger and Food intake, Food intake decreases Hunger by increasing

[1]Note that Crielaard et al.'s (2022) model was slightly modified to correspond with Sterman's recommendations for System Dynamics models. For example, the Stock "Eating" was changed to "Food intake", as it is recommended to use nouns or noun phrases for Stocks.

its outflow rate. It also becomes apparent that Eating triggers is not a Stock but an Auxiliary, as it does not accumulate. This stock-and-flow model will be used to illustrate the functionality of the *sdbuildR* package in the next sections.

The *sdbuildR* package

To build and simulate stock-and-flow models, *sdbuildR* creates an easily modifiable object which contains all the information needed to simulate it. Users can easily add and remove variables, change equations and units Stocks, and incorporate other R functionality. As the simulation script is compiled anew each time, users are alleviated from manually updating the simulation script. The main object built by *sdbuildR* mimics the structure of the XMILE (XML Modeling Interchange LanguagE, (**Eberlein2013?**)) standard. XMILE is the main cross-platform format for System Dynamics models. Most popular System Dynamics software such as Vensim, Stella and Powersim are XMILE-compatible. XMILE uses Extensible Markup Language (XML) format, such that the model is structured using tags such as <stock> (**Bray1997?**). The main top-level tags are "sim_specs" (simulation specifications), "model" (the stock-and-flow model), "model_units" (custom-defined units), and "macro" (global variables). In the model tag, the model's variables are separated into Stocks, Flows, Auxiliaries, and Graphical Functions.

The model is simulated using the *deSolve* package (**deSolve?**), an established package for simulating dynamical systems. Stock-and-flow models are a subset of dynamical systems, such that a dynamical system can be rewritten to an equivalent stock-and-flow model (**sterman2000?**). *deSolve* is written in C and offers a multitude of differential equation solvers, including the standard fixed timestep Runge-Kutta family, as well as a range of adaptive timestep solvers.

Below, we demonstrate how to use *sdbuildR* to build, modify, and simulate stock-and-flow models.

Building a Stock-and-Flow Model

In *sdbuildR*, stock-and-flow models can be created in three ways. Firstly, over XXX example models can be created using `template()`:

```r
sdm = template("Crielaard2022")
```

Secondly, an Insight Maker model can be translated to R using `insightmaker_to_R()`:

```r
URL = "https://insightmaker.com/insight/5LxQr0waZGgBcPJcNTC029/Crielaard-2022"
sdm = insightmaker_to_sdm(URL)
```

And lastly, a stock-and-flow model can be created from scratch. To do so, we first create an XMILE object using `xmile()`, which yields a nested list in the structure of the XMILE standard.

```r
sdm = xmile()
```

Simulation specifications - the start and stop time, the time step, the time units, and the integration method - can be changed via `sim_specs()`:

```r
sdm = sim_specs(sdm,
  method = 'euler', time_units = 'day',
  start = 0, stop = 500, dt = .01
)
```

We then include each variable using `build()`, which takes as the first argument the variable name, the second argument the type of building block ("stock", "flow", "aux", or "gf"), and then any appropriate properties for that type of building block (see Table **??**). Below, we define for both Stocks their initial condition in the *eqn* property, the Flows increasing the Stock in *inflow*, the Flows decreasing the Stock in *outflow*, and an optional *label*. If needed, variable names are translated to syntactically valid and unique R names. For example, "Food intake" would be translated to "Food.intake". Throughout the paper, we make use of the convenient pipe operator `%\>%` from the *magrittr* package (**XXX?**), which simply passes the result of an expression to the next expression as a first argument.

```
sdm = sdm %>%
  build("Food.intake", "stock",
    eqn = "runif(1)",
    inflow = list(c("Feeling.hunger", "Eating.triggers")),
    outflow = list(c("Satiety", "Effect.of.compensatory.behaviour")),
    label = "Food intake"
  ) %>%
  build("Hunger", "stock",
    eqn = "runif(1)",
    inflow = "Losing.energy.by.compensatory.behaviour",
    outflow = "Food.intake.reduces.hunger"
  ) %>%
  build("Compensatory.behaviour", "stock",
    eqn = "runif(1)",
    inflow = "Compensating.for.having.eaten",
    outflow = "Satisfaction.with.hungry.feeling",
    label = "Compensatory behaviour"
  )
```

For each Flow, we define its rate of change in the *eqn* property.

```
sdm = sdm %>%
  build("Losing.energy.by.compensatory.behaviour", "flow",
    eqn = "(0.3 * Compensatory.behaviour) * ((1 - Hunger) / 1)",
    label = "Losing energy by compensatory behaviour"
  ) %>%
  build("Feeling.hunger", "flow",
    eqn = "((0.8 * Hunger^a0) * Food.intake) * ((1 - Food.intake) / 1)",
    label = "Feeling hunger"
  ) %>%
  build("Satiety", "flow",
    eqn = "(1.3 * (Food.intake)) * ((1 - Food.intake) / 1)"
  ) %>%
  build("Food.intake.reduces.hunger", "flow",
    eqn = "((1.4 * Food.intake^a0) * Hunger) * ((1 - Hunger) / 1)",
    label = "Food intake reduces hunger"
  ) %>%
  build("Compensating.for.having.eaten", "flow",
    eqn = "(Sig(a2 * Food.intake)) * ((1 - Compensatory.behaviour) / 1)",
    label = "Compensating for having eaten"
  ) %>%
  build("Satisfaction.with.hungry.feeling", "flow",
    eqn = "(1.3 * Hunger * Compensatory.behaviour) * ((1 - Compensatory.behaviour) / 1)",
    label = "Satisfaction with hungry feeling"
  ) %>%
  build("Eating.triggers", "flow",
    eqn = "(a1 * Food.intake) * ((1 - Food.intake) / 1)",
    label = "Effect of eating triggers"
  ) %>%
  build("Effect.of.compensatory.behaviour", "flow",
    eqn = "(2 * Compensatory.behaviour * Food.intake) * ((1 - Food.intake) / 1)",
    label = "Effect of compensatory behaviour"
  )
```

In this model, all Auxiliaries are constants, and their *eqn* property can thus be passed as a numerical value. Here, we show that variables may also be added in a vectorized manner for efficiency.

```
sdm = sdm %>%
  build(c("a0", "a1", "a2"), "aux", eqn = c(1.31, 1.5, 0.38))
```

Lastly, we define a global variable, i.e. macro, for the sigmoid fuction:

```
sdm = sdm %>% macro(name = "Sig", eqn = "function(x) 1 / (1 + exp(1)^(-x))")
```

Macros may be variables, functions, or any other object.

For any given XMILE object, the code to build the model from scratch can be obtained with `get_build_code()`.

```
get_build_code(template("SIR"))
```

Table 2: Properties for each Building Block

Property	Description	Example	Stock	Flow	Auxiliary	GF
label	Name for plotting	`'Food intake'`	x	x	x	x
eqn	Equation (initial value in case of Stock)	`'sqrt(10)'`	x	x	x	
units	Units	`'second'`	x	x	x	x
non_negativity	Enforce non-negativity	`FALSE`	x	x	x	
min	Minimum value constraint	`0`	x	x	x	x
max	Maximum value constraint	`300`	x	x	x	x
inflow	Variable flowing into Stock	`'Hunger'`	x			
outflow	Variable flowing from Stock	`'Hunger'`	x			
to	Target variable of Flow	`'Hunger'`		x		
from	Source variable of Flow	`'Hunger'`		x		
xpts	x-domain points of GF	`c(0, 1, 2, 3)`				x
ypts	y-domain points of GF	`c(10, 12, 14, 16)`				x
interpolation	Interpolation method of GF	`'linear'`				x
rule	Rule determining behaviour of GF outside of x-domain	`2`				x
note	Add documentation to variable	`'Subjectively assessed hunger on a scale from 0 to 10'`	x	x	x	x

Variable properties can be modified by using `build()` without the building block. In this example, the initial value of Food intake is changed:

```r
sdm = build(sdm, "Food.intake", eqn = .5)
```

To view any variable properties, use `peek()`:

```r
peek(sdm, "Food.intake", "eqn")
```

Variables can be removed using `erase()`:

```r
sdm = erase(sdm, name = "Food.intake")
```

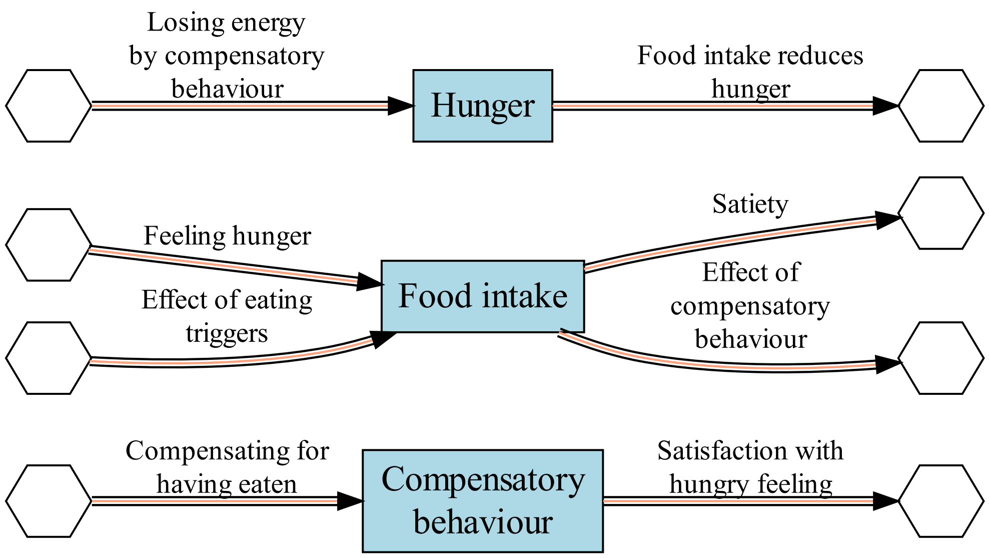
An overview of the variables in the model can be obtained with `summary()`:

```r
summary(sdm)
```

```
## Your model contains:
## 3 Stocks: Food.intake, Hunger, Compensatory.behaviour
## 8 Flows: Losing.energy.by.compensatory.behaviour, Feeling.hunger, Satiety, Food.intake.reduces.hunge
## 3 Auxiliaries: a0, a1, a2
## 0 Graphical Functions
```

A model diagram of the Stocks and Flows can be plotted via the *DiagrammeR* package. Stocks are indicated as vertices, Flows are indicated by arrows entering (inflows) or leaving (outflows) Stocks, and sources and sinks of Flows are indicated by open hexagons (equivalent to clouds in traditional SD diagrams).

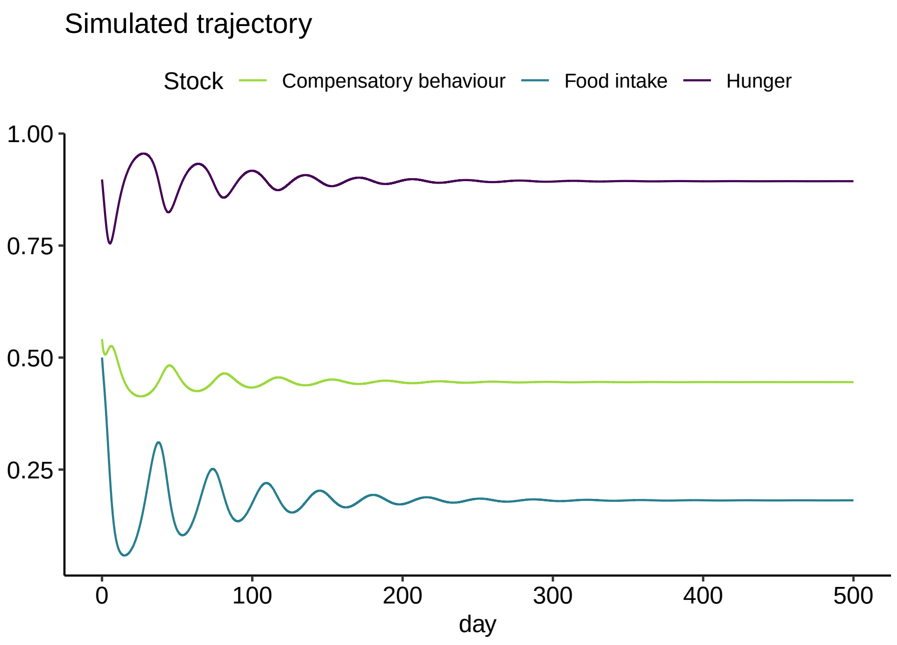
```r
plot(sdm)
```



Losing energy by compensatory behaviour
Hunger
Food intake reduces hunger
Feeling hunger
Satiety
Food intake
Effect of eating triggers
Effect of compensatory behaviour
Compensating for having eaten
Compensatory behaviour
Satisfaction with hungry feeling

To simulate the model, simply run `simulate()`. The timeseries of the Stocks can be plotted with `plot_stocks()`.

```r
sim = simulate(sdm)
plot_stocks(sdm, sim$df)
```



The simulation script that is created can also be accessed by the user via `compile()`, which yields a complete, annotated R script. The R script includes the necessary libraries, variable and function definitions, a call to *deSolve* to simulate the ODE, and a basic time series plot (see Section ??). An example of the output of `compile()` may be found in the Appendix.

Table 3: Main functions in *sdbuildR*

Function	Purpose
`xmile()`	Create object of class xmile
`build()`	Add variable to model
`plot()`	Plot model diagram
`summary()`	Print summary of building blocks and variables
`simulate()`	Simulate model
`plot_stocks()`	Plot timeseries of Stocks
`compile()`	Get code to simulate model
`peek()`	View desired variable properties
`sim_specs()`	Modify simulation specifications
`model_units()`	Define custom units
`macro()`	Define global variables
`header()`	Define header of model
`get_build_code()`	Get code to build model from scratch
`template()`	Access template models
`insightmaker_to_R()`	Translate Insight Maker model to xmile object

Model constraints

System Dynamics models can typically have three types of constraints: non-negative Flows, non-negative Stocks, and minimum and maximum value constraints. When a Flow is constrained to be non-negative, it will only ever be zero or positive by setting it to zero when negative. Similarly, non-negative Stocks always remain zero or positive by setting the update in state to zero when the net flow is negative. Finally, minimum and maximum value constraints allow the user to specify the range of values a variable may have. When this latter constraint is violated, the simulation stops.

To implement non-negativity of Stocks and Flows, *sdbuildR* constrains Flows and the change in Stocks in the ODE function. Specifically, Flows are kept non-negative using the `nonnegative()` function, which outputs 0 in case Flows are negative (preserving units in case these are specified). Stocks can be constrained to only zero or positive values by making use of the root function in *deSolve*. The root function is triggered when selected Stocks go below or exceed a certain threshold. A corresponding event function specifies the change in the state variables when the root function is triggered. To enable non-negative Stocks, the root function is triggered when selected Stocks go below zero, and are set to zero in the event function.

However, including such hard-coded constraints in the model is not recommended (**sterman2000?**). If a Stock or Flow can logically never be negative, for example in the case of animals or deaths, this should ideally arise from the model equations and parameters itself. If they unintentionally turn negative, it is likely due to model misspecification and including hard-coded constraints can lead to unexpected results (**sterman2000?**). To incorporate checks for when minimal and maximum values are crossed, each variable may have a *min* and *max* property:

```r
sdm = build(sdm, "Food.intake", min = 0, max = 1) %>%
  build(sdm, "Losing.energy.by.compensatory.behaviour", max = 1)
```

When compiling the R script, minimum and maximum value constraints are created using `get_constraint_logical()`, which takes as input a list of minimum and maximum values. For example,

```r
# Constraints of minimum and maximum value
constraints_def = list(
  "Food.intake" = c(min = 0, max = 1),
  "Losing.energy.by.compensatory.behaviour" = c(max = 1)
)
constraints = get_logical_constraints(constraints_def)
print(constraints)
```

```
## [1] "drop_if_units(Food.intake) < 0"
## [2] "drop_if_units(Food.intake) > 1"
## [3] "drop_if_units(Losing.energy.by.compensatory.behaviour) > 1"
```

which outputs a vector of strings with constraints to evaluate. In the ODE, these constraints are checked using `check_constraints()`.

```r
check_constraints(constraints, environment(), t)
```

where `environment()` provides access to the variables computed in the ODE. When violated, the simulation stops, with a message

```
## Constraint violated at time 1
## drop_if_units(Food.intake) < 0: FALSE
## drop_if_units(Food.intake) > 1: TRUE
## drop_if_units(Losing.energy.by.compensatory.behaviour) > 1: FALSE
```

```
##

## Error in check_constraints(constraints, envir, 1):
```

Interpolation functions

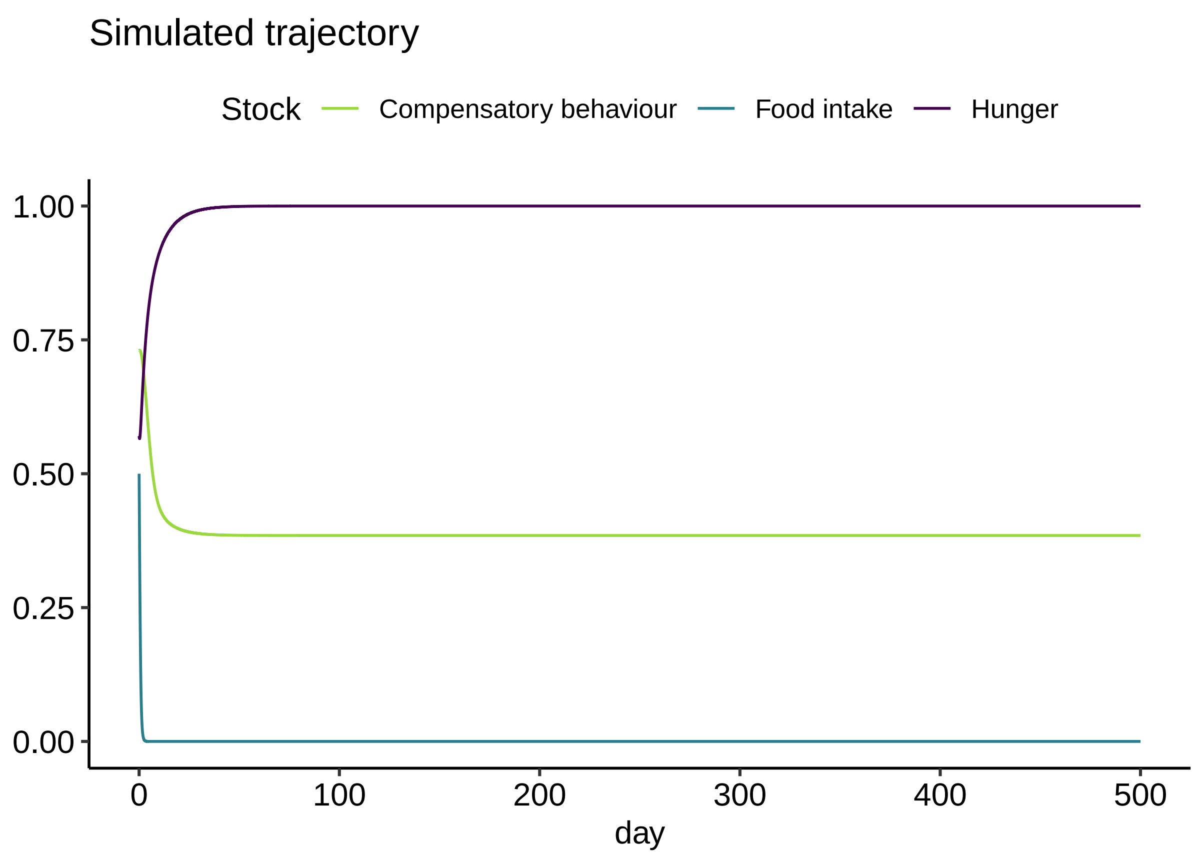
In System Dynamics modelling, it is common to incorporate time-dependent exogenous inputs, which can represent various external influences on the system. Among these inputs are *step* functions for sudden, sustained level changes, *ramp* functions for gradual increases or decreases, *pulse* functions for instantaneous bursts, and *seasonal* functions for cyclical patterns. These functions can be created as an interpolation function using `approxfun()`, which underlies the convenience functions offered by *sdbuildR* to model these input functions:

```
times = seq(0, 50, by = .01)
step = make_step(times, start_t_step = 10, h_step = 20)
ramp = make_ramp(times, start_t_ramp = 10, end_t_ramp = 20)
pulse = make_pulse(times,
  start_t_pulse = 10, h_pulse = 10,
  w_pulse = 5, repeat_interval = 20
)
seasonal = make_seasonal(times, peak_time = 1, 'years')
```

As an example, to model a regular pulse in eating triggers, we can create a pulse function named "trigger" which repeats every day (24 hours). This function can then be included in other equations, and here is indexed with the current time step t:

```
sdm = sdm %>%
  build("ramp", "aux", eqn = "make_ramp(times, start_t_ramp = 0, end_t_ramp = 500,
                  start_h_ramp = 1, end_h_ramp = 1)") %>%
  build("Eating.triggers", eqn = "(a1 * ramp(t) * Food.intake) * ((1 - Food.intake) / 1)")

sim = simulate(sdm)
plot_stocks(sdm, sim$df)
```



Similarly, a Graphical Function is also an interpolation function. With a Graphical Function, the user can create a function of any shape by giving a set of points on the x-domain and y-domain:

```
sdm = sdm %>%
  erase("ramp") %>%
  build("ramp", "gf",
    xpts = c(0, 300, 350, 400, 500),
    ypts = c(.1, .5, .2, .5, 1.5)
  )

sim = simulate(sdm)
plot_stocks(sdm, sim$df)
```

Delays, Smoothing, and Past Dependencies

Not all interactions in a system occur instantaneously. When the output of a variable lags behind its input, this is called a *delay* (Sterman, p. 411). A delay may be *fixed* (i.e. a pipeline delay) or *smooth* (i.e. first-order and higher-order delays). In the case of fixed delays, the input is delayed for a fixed delay length τ and arrives all at once. For example, material on a conveyor belt arrives at its destination after a fixed amount of time, without any mixing of materials. However, in other systems, the arrival of delayed material or information does not occur at a fixed time, but is rather a smooth process. The variability in delay time and arrival requires a smooth delay function. The simplest delay is a first-order delay function, in which an intermediate Stock (the accumulator) accrues input and releases it at a rate of $1/\tau$. This allows for intermediate mixing of material and a smooth delayed arrival of the input, as opposed to an abrupt release of all material. A first-order delay is suitable for processes with one intermediate step. When there are multiple stages in the delay process, a higher-order delay structure may be used. For example, there are multiple time delays between investment in innovation and the output of that innovation. The higher the order of the delay, the smoother the output. Higher-order delays are cascades of first-order delays, and need as many accumulators as the number of orders. Each stage is assumed to have the same time delay.

To facilitate modelling a fixed or smooth delay, *sdbuildR* sets up the necessary components of a delay structure, relieving the user from specifying this themselves. In brief, a fixed delay can be included by specifying `add_delay()` in the equation. As a first argument, the variable that is delayed is specified, followed by the length of the delay, with as an optional argument the default value when the simulation has not reached the length of the delay yet.

```r
sdm = sdm %>%
  build("Satiety",
    eqn = "(1.3 * add_delay(Food.intake, 1)) * ((1 - Food.intake) / 1)"
  )

sim = simulate(sdm)
plot_stocks(sdm, sim$df)
```

Note that the delay length is specified in time units, not indices, meaning a delay length of 1 with a simulation time unit “second” refers to one second in the past. Similarly, a *n*th-order delay can be included using `add_delayN()`, which takes as its first argument the variable that is delayed, as a second argument the length of the delay, as a third argument the order of the delay, and as an optional fourth argument the default value when simulation has not reached the length of the delay yet.

```r
sdm = build(sdm, "Food.intake.reduces.hunger",
  eqn = "((1.4 * add_delayN(Food.intake, 1, 50)^a0) * Hunger) * ((1 - Hunger) / 1)"
)
sim = simulate(sdm)
plot_stocks(sdm, sim$df)
```

Similar to a fixed delay, users may also access past values in a particular interval using `add_past()`. `add_past()` takes as its first argument the variable to access past values of, and as an optional second argument the interval to retrieve from the current time point. For example, a past interval of 10 retrieves at time 100 returns past values from time 90 to time 100 (inclusive). When the past interval is omitted, all past values from the current time point are returned. Here, we let Compensatory behaviour be influenced by the maximum value of Food intake over the last four hours.

```r
sdm = sdm %>%
  build("Compensating.for.having.eaten",
    eqn = "(Sig(a2 * max(add_past(Food.intake, 4)))) * ((1 - Compensatory.behaviour) / 1)"
  )

sim = simulate(sdm)
plot_stocks(sdm, sim$df)
```

XXX graphical overview of delay, delayN, past, conveyor, smooth

Function Name	Purpose	Arguments
add_delay	Add a fixed delay	Variable, delay length, (default value)

Function Name	Purpose	Arguments
add_delayN	Add a smooth delay of order N	Variable, delay length, delay order, (default value)
add_past	Retrieve past values	Variable, (past interval)

We briefly elaborate on how *sdbuildR* sets up the necessary components for delay structures and retrieving past values when compiling the R script or simulating the model. In the case of a fixed delay, access to a variable's history is needed, which is not provided by deSolve in all cases. In *deSolve*, past values can only be accessed using the `lagvalue()` and `lagderiv()` function when running a delay differential equation with `dede()`. These only give access to past values of state variables and derivatives, and the Euler and Runge-Kutta 4th order solvers are not available for `dede()`. To be able to access past values of all variables using the "euler" and "rk4" integration solver, *sdbuildR* uses a different solution. In the case of fixed delays, a global dataframe *archive* is created, which keeps track of all desired variables.

```r
arhive_var = c("Food.intake")
archive <<- setup_archive(archive_var, times)
```

In the ODE, the archive is updated in each step using `update_archive()`, which requires the name of the global dataframe, the current timestep, and the current environment in the ODE. This convenience function allows the ODE function to remain uncluttered by individual updates for each variable, updating the archive in one sweep.

```r
update_archive("archive", t, environment())
```

To retrieve values from the archive, we can use either `delay()` (to retrieve a single value) or `past()` (to retrieve an interval of values). As in Insight Maker, values are linearly interpolated to deal with any numerical inaccuracies in the current time value t and to make the method suitable for the "rk4" solver. Notably, accessing past values is only possible when using either "euler" or "rk4" integration, not with solver methods which use variable time step sizes.

```r
past(archive, t, "Food.intake", pars$past_interval)
delay(archive, t, "Food.intake", pars$delay_length)
```

This also illustrates that any specified parameters (length and order of delay, interval of past) are added to the model as Auxiliaries.

In the case of smooth delays, a delay accumulator chain needs to be included as a Stock. The number of links in the chain is the order of the delay, such that a 10th order delay requires 10 additional Stocks. Rather than writing an individual equation for each accumulator, *sdbuildR* sets up all accumulators using `setup_delayN()`:

```r
xstart = c(xstart, setup_delayN(
  "Food.intake_delayN_acc", xstart$Food.intake,
  pars$length_delayN, pars$order_delayN
))
```

This allows the users to easily adjust the order of the delay. In the ODE, a separate variable is created to update the delay chain and to access the output of the last component in the chain.

```
Food.intake_delayN = delayN(
  input, S[grep('^Food.intake_delayN_acc', names(S))],
  pars$length_delay, pars$order_delay
)
```

The flow from the last accumulator in the delay chain can be accessed with the *outflow* entry (`Food.intake_delayN$outflow`). Accumulators in the delay chain are updated using the *update* entry (`Food.intake_delayN$update`), which is of the same length as the number of orders in the chain.

Units

Units in System Dynamics models serve two purposes: model verification and model flexibility. Units, such as meters, seconds, or degrees Fahrenheit, help the user check for logical consistency within their model. For instance, Flows need to have the units of the Stock they are connected to divided by a time unit. This means that any Flows connect to Food intake has to have the units kilocalories divided by a time unit such as hour or day. Similarly, mathematical operations between incompatible units are not allowed: *Satiety* (units Kilocalories/hour) cannot be subtracted from *Food.intake.reduces.hunger* (units Hunger intensity/hour). As such, specifying units forces users to think about the meaning of their variables and their equations.

Units also allow for greater flexibility in how each model element is defined. System Dynamics software automatically converts between units, such that variables may be defined on different levels (e.g. seconds and minutes). An error is thrown in case units are incompatible, which provides a consistent model verification check.

To implement units, *sdbuildR* uses the *units* package, which allows for flexible, comprehensive use of units. For example, to create a variable *a* of value 2 with units seconds:

```
a = set_units(2, 'second')
```

Units of a variable may be specified via:

```
sdm = build(sdm, "Food intake", units = "Kilocalories")
```

All available units in the units package can be printed in a table using `units::valid_udunits()`, listing over 250 units. This table includes the definition of each unit in terms of other units. Moreover, each unit can be combined with a prefix, such as “nano”. All possible prefixes are listed in `units::valid_udunits_prefixes()`. Some units, such as “year”, may have an unexpected definition. Rather than 365 days, the unit “year” in the units package accounts for leap years, making it approximately 365.24 days. The unit “common_year” is 365 days. To check a unit’s definition, simply convert it using `set_units()` to the desired unit.

```
set_units(set_units(1, "year"), "day")
```

```
## 365.2422 [d]
```

```
set_units(set_units(1, "common_year"), "day")
```

```
## 365 [d]
```

```
set_units(set_units(1, "month"), "day")
```

```
## 30.43685 [d]
```

If these pre-defined units are insufficient, users can define their own custom units. In the XMILE format, there is a separate tag for custom units. To add these, use `model_units()` with as a first argument the symbol of the unit, as an optional second argument the definition of the unit in terms of other units, and as an optional third argument an alternative name. No definition or an empty definition creates a new base unit: a unit not defined in terms of other units. Unit symbols and names may be modified to create syntactically valid names. In this case, "Satiety Hunger Score" is changed to "Satiety_Hunger_Score".

```
sdm = model_units(sdm, name = "Satiety Hunger Score", eqn = "300 kilocalories", alias = "SHS") %>%
  build("Hunger", units = "SHS")
```

If any units are used in the R script, the *units* package is loaded at the top of the R script with:

```
# Add package for specifying units of each model element
if (!require('units')) install.packages('units')
library(units)
units_options(allow_mixed = T, simplify = T, set_units_mode = 'standard')
```

These options ensure that vectors can contain variables with different units (*allow_mixed*), that units are reduced to the most simple expression (*simplify*), and that variables can be used to set units, without being interpreted as the unit itself (*set_units_mode*, e.g. `a = 'second'; set_units(1, a)`).

Any undefined units are installed with *sdbuildR*'s wrapper function `install_custom_units()`, which ensures any existing units of the same name are removed before installation.

Any variable without specified units is ensured to not have units using `drop_if_units()`.

On a final note, though specifying units helps with model verification, calling functions from the units package in the ODE slows down the computation considerably. Users interested in efficient simulations are recommended to ensure unit compatibility manually, without using the units package in the ODE. All unit translation can be turned off by calling `compile()` or `simulate()` with `keep_unit = FALSE`, though this is set to TRUE by default to ensure the model is reproduced accurately.

Implementation of `compile()`

The function `compile()` creates an annotated, stand-alone script to simulate a stock-and-flow model. It calls the necessary libraries and defines the timing sequence, any optional custom-defined units, the ODE function, an optional root and event function in case of non-negative Stocks, static parameters, the initial condition, a call to the ODE solver to simulate the timeseries, and a basic plot of the timeseries. In order to do so, `compile()` executes the following:

1. Any fixed or nth order delays, smoothing, or reference to past values are detected and the necessary components are assembled (see Section **??**);
2. Auxiliaries are divided into static parameters or dynamically updated variables in the ODE based on their dependencies;
3. Topological ordering is applied to both the static parameters and initial conditions as well as the ODE;
4. Any incompatible unit operations are attempted to be fixed (see section Units **??**);
5. Reference to static parameters or initial conditions outside of the ODE are prefixed with 'pars'*or*'*xstart*', respectively.

XXX include figure with workflow

Topological ordering ensures equations are ordered correctly according to their dependencies. For instance, if the initial value of Hunger depends on Food intake, then the initial value of Food intake should be defined before that of Hunger. All variables may be defined as a function of one another, as long as there are no circularities. For instance, the initial value of Food intake may not be defined as the initial value of Hunger:

To define variables in the correct order, the dependencies of each variable are transformed into graph "edges", where an edge "xstart$Hunger, xstart$Food.intake" means that xstart$Hunger needs to be defined before xstart$Food.intake These edges are then fed to `topo_sort()` from the *igraph* package, which yields a correct ordering of variables (though there may be multiple ways to order them).

Validation: Insight Maker to R

To ensure stock-and-flow functionality was implemented correctly, *sdbuildR* has been validated using Insight Maker. Insight Maker is a free and open-source web-based tool (https://insightmaker.com/) to build, simulate, and optimize stock-and-flow models and agent-based Models, as well as draw Causal Loop Diagrams. InsightMaker runs on JavaScript supported by the *simulation* (openly available at https://github.com/scottfr/simulation/). Insight Maker includes extensive features to help communicate model structure and behaviour, such as scenarios and interactive exploration of parameter settings, as well as simple sensitivity testing and parameter optimization. Insight Maker is used among others for modelling health care, environment, ecological and biological systems, as well as for educational purposes. A handful of other free software exists (see https://en.wikipedia.org/wiki/Comparison_of_system_dynamics_software; https://github.com/SDXorg/SD-Tools), but Insight Maker stands out with its intuitive user interface and tutorial guidance, which makes modelling accessible to users without a technical background.

Though Insight Maker is highly accessible and powerful, users may want to export their Insight Maker model to R for the same reasons outlined in the Introduction: accessibility, compatibility, flexibility, scalability, and reproducibility. Though all these desiderata are supported by Java, this may not be the language users are versed in. For R users, an Insight Maker model in R enables more advanced System Dynamics techniques, or which they can use already existing R packages. For example, different sampling designs for parameter and structural sensitivity testing can easily be executed in R, such as Latin hypercube or Monte Carlo sampling (**XXX?**). Implementing these analyses in R would also allow users to fully utilize the computational power they have access to. A fully reproducible analysis script will add robustness to the model, adding confidence to model results.

InsightMaker offers 126 built-in functions relating to basic mathematical, vector and string operations, random number generators and statistical distributions, accessing past values, time conversion functions, and user input functions. To allow for greater flexibility, users can also define their own global variables. To ensure *sdbuildR* reproduced Insight Maker functionality, over a hundred stock-and-flow models were plucked from the community models available on Insight Maker. Each URL was translated to an .InsightMaker file. The models were simulated using the *simulation* package in JavaScript,**??** as well as in R after translation using `insightmaker_to_R()`. Model output was reproduced up until the fifth decimal, as shown in Supplementary Figure **??**. The scripts necessary to reproduce the validation may be found in the folder sdbuildR/validate.

The translation of `insightmaker_to_R()` has three limitations. Firstly, the non-negative Stock feature is not reproduced in *sdbuildR* (see section **??**). *sdbuildR* uses *deSolve*'s root function to ensure Stocks remain non-negative, whereas Insight Maker adjusts outflows to maintain non-negativity. In simple cases, these approaches produce the same output, but for models where non-negative Stocks have multiple inflows and outflows, they do not. In the validation, all models have been adjusted to turn off the non-negative feature of Stocks. The second limitation concerns propagated double precision differences. Double precision floating point math engines (using the standard) have small inaccuracies in their representation of numbers. Though trivial, these differences can propagate throughout the simulation to create large differences. Models in which this occurred were excluded from the validation. The final limitation relates to the difference between the *simulation* package and *deSolve*. *deSolve* expects the initial condition, parameters, and ODE function to be specified separately. Conversely, Insight Maker allows the initial condition to be defined by dynamically computed variables. Models were excluded in case it was not possible to pluck this dynamically computed variable from the ODE to define the initial condition because of topological ordering impossibilities. For further details and implementation of the validation, please see the Appendix.

Limitations

XXX

Conclusion

The package *sdbuildR* enables building and simulating stock-and-flow models in R, with the aim of making System Dynamics modelling accessible and flexible to fit the user's needs. *sdbuildR* supports the use of model constraints, constructing interpolation functions, implementing (*n*th order) delays, smoothing and retrieving past values, specifying (custom) units as well as global variables. As the main object is formatted in XMILE-style, it promotes coherence with other System Dynamics software. By translating Insight Maker models, *sdbuildR* facilitates flexible and advanced use of System Dynamics modelling.

Providing the basic building blocks of stock-and-flow models is just the first step, however. System Dynamics modelling is an iterative, hands-on process, which involves defining a problem, formulating a dynamic hypothesis, formalizing a model, testing, and validation (**sterman2000?**). Fortunately, many subcomponents of this process are already supported in R. Data wrangling needed both for empirical data and simulation results is well-supported by the *tidyverse* libraries (for a tutorial in the context of System Dynamics, see (**duggan2018?**)). Interactive simulations could be facilitated using a Shiny app to mimick the interface of Insight Maker (**Shiny?**). For model testing, Duggan (2017) illustrates how the R package *RUnit* can be used to validate the model (**duggan2017?**). Model validation includes structural validity - whether the system's structure conceptually and numerically corresponds to the real world - and behavioural validity - whether simulated system behaviour matches real-world behaviour (**duggan2017?**). To identify the most influential parameters in a model, Duggan outlines how to implement statistical screening in R (**duggan2017?**).

In addition to already existing tools, the integration and development of new tools for System Dynamics becomes possible. The Python package *pysd* is an excellent demonstration of new possibilities that emerge when System Dynamics is implemented in an open-access coding language. By drawing on functionality from other packages, *pysd* enables the estimation of model parameters from empirical data using Markov-chain Monte Carlo simulation (MCMC), or even the estimation of the entire functional form of an equation using machine learning. Systems can be simulated in real-time with animated figures to better communicate the behaviour of the system over time. Support and guidance for such analyses could also be offered in R, and possible even extended. For example, AI-aided modelling can already be used to create Causal Loop Diagrams using the R package *theoraizer* (**Waaijers2024?**).

Though the efficiency and analytical abilities of licensed System Dynamics software are vast, its proprietary nature makes is naturally less accessible and less able to integrate new research and tools. For example, as of yet, artificial intelligence supported modelling is not implemented in Stella, Vensim, or Powersim, whereas *pysd* has offered this since 2017. In the same vein, *sdbuildR* is an effort to encourage the R community to engage in a collaborative effort to utilize and advance System Dynamics modelling.

Appendix

Example compiled R script

```
compile(sdm)
```

```
## # Script generated on 2024-11-28 11:25:57.510613 by sdbuildR. Please cite ***
##
## # Load packages
## library(dplyr)
## library(ggplot2)
## library(insightmakeR1)
##
##
##
## # Define time sequence
## dt <- 0.01
## times <- seq(from = 0, to = 500, by = dt)
##
## # Simulation time unit (smallest time scale in your model)
## time_units <- "day"
##
##
## Sig <- function(x) 1 / (1 + exp(1)^(-x))
##
##
##
##
##
## # Define ODE
## ode_func <- function(t, S, pars) {
##   S <- as.list(S)
##
##
##   # Compute change in Stocks at current time t
##   with(c(S, pars), {
##     # Get names of variables in environment
##     env_var <- names(environment())
##
##     # Update Auxiliaries and Flows
##     Losing.energy.by.compensatory.behaviour <- (0.3 * Compensatory.behaviour) * ((1 - Hunger) / 1) #
##     Feeling.hunger <- ((0.8 * Hunger^a0) * Food.intake) * ((1 - Food.intake) / 1) # Flow to Food.inta
##     Satiety <- (1.3 * (Food.intake)) * ((1 - Food.intake) / 1) # Flow from Food.intake
##     Food.intake.reduces.hunger <- ((1.4 * Food.intake^a0) * Hunger) * ((1 - Hunger) / 1) # Flow from
##     Compensating.for.having.eaten <- (Sig(a2 * Food.intake)) * ((1 - Compensatory.behaviour) / 1) # 
##     Satisfaction.with.hungry.feeling <- (1.3 * Hunger * Compensatory.behaviour) * ((1 - Compensatory
##     Eating.triggers <- (a1 * ramp(t) * Food.intake) * ((1 - Food.intake) / 1) # Flow to Food.intake
##     Effect.of.compensatory.behaviour <- (2 * Compensatory.behaviour * Food.intake) * ((1 - Food.intal
##
##     # Collect inflows and outflows for each Stock
##     dCompensatory.behaviour <- Compensating.for.having.eaten - Satisfaction.with.hungry.feeling
##     dFood.intake <- Feeling.hunger + Eating.triggers - Satiety - Effect.of.compensatory.behaviour
##     dHunger <- Losing.energy.by.compensatory.behaviour - Food.intake.reduces.hunger
##
##
##     # Collect all newly created variables
##     env_update <- collect_var(environment(), setdiff(names(environment()), c("env_var", env_var)))
##
##     # Combine change in Stocks
##     dSdt <- c(dCompensatory.behaviour, dFood.intake, dHunger)
##
##     return(list(dSdt, env_update))
##   })
## }
##
## # Set-up parameters and initial condition
## pars <- list()
## xstart <- list()
##
## # Define parameters, initial conditions, and functions in correct order
## pars$a0 <- 1.31
## pars$a1 <- 1.5
## pars$a2 <- 0.38
## pars$ramp <- make_ramp(times, start_t_ramp = 0, end_t_ramp = 500, start_h_ramp = 1, end_h_ramp = 1)
## xstart$Food.intake <- 0.5
## xstart$Hunger <- runif(1)
## xstart$Compensatory.behaviour <- runif(1)
##
## # Turn initial condition into alphabetically ordered named vector
## xstart <- unlist(xstart)[order(names(xstart))]
##
## # Run ODE
## out <- deSolve::ode(
##   func = ode_func,
##   y = xstart,
##   times = times,
##   parms = pars,
##   method = "euler"
## )
## df <- as.data.frame(out)
##
## # Plot ODE
## plot_stocks(sdm, df)
```

In general, *deSolve*'s `ode()` only outputs state variables (i.e. Stocks). In order to generate a complete dataframe with all computed variables in the ODE, all created variables are included using `collect_var()`.