3Worlds user manual

Jacques Gignoux – jacques.gignoux@upmc.fr · Ian D. Davies – ian.davies@anu.edu.au · Shayne R. Flint – shayne.flint@anu.edu.au – Version 1.5, 29 April 2021 |

Table of Contents

---

by **Jacques Gignoux**, **Ian D. Davies** & **Shayne R. Flint**

Version: **1.5** (29 April 2021)

1. Purpose and motivation

1.1. Why another simulation environment for ecology?

"Though the organisms may claim our primary interest, when we are trying to think fundamentally we cannot separate them from their special environment, with which they form one physical system" A. G. Tanlsey (1935).

Ecosystems, as first proposed by Arthur Tansley (1935), have both physical and biological aspects. They comprise not only energy and chemical stocks and flows but also living things that are born, reproduce and die, exhibiting particular behaviours over the course of their existence. They are made from and return to the physical world, are part of it and at the same time, constitute a domain where terms such as reproduction, social interaction, communication and life cycle have meaning.

To this day, reconciling these two aspects, where the physical is articulated in SI units and the biological in descriptive terms (e. g. heterotroph, shade-tolerant, obligate seeder), remains a key problem in ecosystem studies: How does an energy flux translate into a number of living individuals or in a higher biodiversity? How might a particular reproduction strategy affect river hydrology and the regional water budget (e. g. the beaver)? How do social interactions affect the frequency of hurricanes (e. g. climate change)? **3Worlds** aims at providing *a tool for building simulations that couple the physical and biological aspects of ecosystems*.

It is our view that existing tools focus predominantly on one or other of these aspects. For example, dynamic models based on differential equations and can only represent reproduction and growth using metaphors for continuous physical quantities (e.g. matrix population models). Multi-agent systems (MAS), on the other hand, model social interactions well but pose difficulties in balancing mass or energy budgets.

In addition to this physical and biological duality, ecosystems are '*multi-scale*' (cf. an extensive discussion on this topic and other questions related to the ecosystem concept in Gignoux et al. 2011). Tansley's ecosystem definition applies equally well to the whole biosphere as to a single bacteria in a drop of water. Currently, there is no software enabling one to consider ecosystems at such different spatial and temporal scales within the same conceptual framework. A second aim of **3Worlds** is to fill this gap. If the ecosystem concept is robust enough, and we believe it is given its success in ecology and its adoption in other scientific fields, then it should be possible to design a conceptually sound and consistent computing framework to deal with a multiplicity of scales.

However, changing the scale of a model is not just a matter of changing spatio-temporal resolution. It also involves the level of detail of the representations used in the model, something which has been formalized in *abstraction theory* (Zucker 2003): the more detailed a model, the less abstract it is. The question arises then as to what level of detail should we model our system to capture its important behaviour? How does changing scale (in this broadest sense) effect model outcomes? There can be no general answer to this question beyond trial and error. This makes the possibility to test and compare different representations of the same system at different scales an essential feature of any software dealing with ecosystems. **3Worlds** provides *great flexibility in ecosystem representation* enabling one to represent *any* ecosystem at *any* spatial and temporal scale with *any* level of detail — all this being determined by the researcher in accordance with the purpose of their study. The researcher has full control on the detailed biological functions and variables relevant to their study.

How is this flexibility compatible with the strong conceptual backbone? **3Worlds** uses *aspect-oriented thinking (AOT)* (Flint 2006), a method to build complex systems (e.g. software) from independent areas of knowledge. We have designed an *archetype* of what we believe constitutes an ecosystem — a recursive and multi-scale system of interacting entities. We use an implementation of AOT to ensure that any user-defined representation of an ecosystem complies with the archetype. This guarantees that, despite a great freedom left to the modeller, their model will always be compatible with the 3Worlds software. The great benefit of this is that while we believe it's possible to construct any type of model within this archetype, imposing specification constraints greatly assists in model comparison: why should two models, ostensibly constructed for the same purpose, differ in their outputs? How does a change in temporal or spatial scale affect projections? How does adding or removing sub-systems change model projections? *Models developed in **3Worlds** are always comparable*, unlike many large model intercomparison exercises where the models are independently developed without reference to any common structuring principals.

The ecosystem is a recursive concept (Gignoux et al. 2011): ecosystems can be nested. Parts of an ecosystem can be studied as ecosystems themselves. We use the concept of a hierarchical system, formalized as a graph (Gignoux et al. 2017) to implement this idea. In **3Worlds**, *the modelled ecosystem is a graph* where nodes represent relevant entities: individuals, populations, climate, soil, area, …; and edges represent relations between these entities. These relations can be of any kind, including a hierarchical relation describing the complex nesting of sub-systems. This provides an elegant solution to the apparent complexity of ecosystems: it allows for various types of *emergence*, enables the comparison of system structures and simulation trajectories, and can represent virtually anything an ecological modeller can conceive of.

3Worlds builds upon more than thirty years of experience of its developers in ecosystem and complex system modelling and simulation. Over this time, we have reached the conclusion that: (1) robust concepts are fundamental for building sound software; (2) some problems, that appear over and over again in the life of a simulation modeller, often have well-established solutions, sometimes in other scientific fields; and (3) complex systems are more easily managed and understood when designed within a sound framework.

Many of the ideas we use are already available, it's just a matter of making them work together, and **3Worlds** is our best attempt to do so!

1.2. Design Concepts

3Worlds is based on a few concepts and techniques that have a very broad application and harnesses them to fit the needs of ecosystem modelling.

1.2.1. Individual based models

Fundamentally, **3Worlds** is designed as a framework for *individual-based models* (IBMs: Grimm & Railsback 2005). IBMs, like multi-agent systems (MASs: Bousquet & Lepage 2004), assume that some important properties of complex systems (like ecosystems and human societies) cannot be understood without representing the detailed behaviour of every individual in the system. In an IBM, a list of 'individuals' (called 'agents' in MASs) is kept in memory, and their interactions through various functions yield the dynamics of the whole system over time. Individuals can come and go, which means that their list is constantly changing in size.

3Worlds simulates a *system* made of interacting *system components*, which are equivalent to the individuals or agents of IBMs and MASs. If required by a simple application, the 3Worlds *system* can be reduced to zero components, just as in any system dynamics application; but this is not the use 3Worlds has been designed for, and other more specialised software may be more adapted for this use case.

1.2.2. The complex system as a dynamic graph

Ecosystems are commonly considered as *complex systems*, without much agreement on what this concept means. Complex systems are usually assumed to possess *emergent properties*; again, with little agreement on what this means. We argued in Gignoux et al. (2017) that the only common feature of all definitions of emergence is that emergence can only appear in systems with a 'macro-state' and 'micro-states'. We called such systems *hierarchical systems* and assumed that they can be ideally represented with a *dynamic graph*.

According to graph theory (https://en.wikipedia.org/wiki/Graph_theory), a *graph* is a mathematical construct comprising a set of *nodes*, a set of *edges* linking these nodes defined in an *incidence function* (Figure 1). A *dynamic* graph (Harary & Gupta 1997) is a graph where these three components (the sets of nodes, edges, and the incidence function) can change over time.

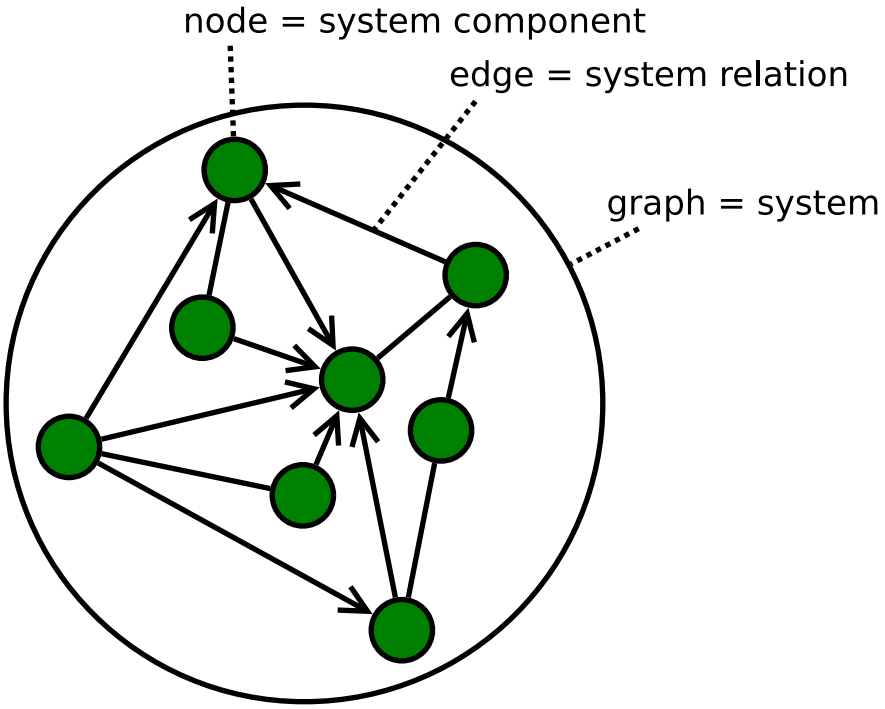


Figure 1. A mathematical graph, showing both the usual mathematical terms and the vocabulary used in 3Worlds. The

incidence function defines edges between nodes. Edges can be directed (with arrows) or undirected.

With the addition of *descriptors* (e.g. numbers with a specific meaning: Figure 2) to nodes and edges, we can use the graph nodes as the individuals of our IBM, and the graph edges give a concrete existence to the interactions between those nodes. If we define *functions* that can modify nodes and edges and their descriptors, the graph becomes dynamic. A **3Worlds** simulation runs a dynamic graph where nodes, edges, descriptors, functions can be adapted to the user's particular application. Nodes are called *system components*, edges *system relations*, and descriptors *properties*.

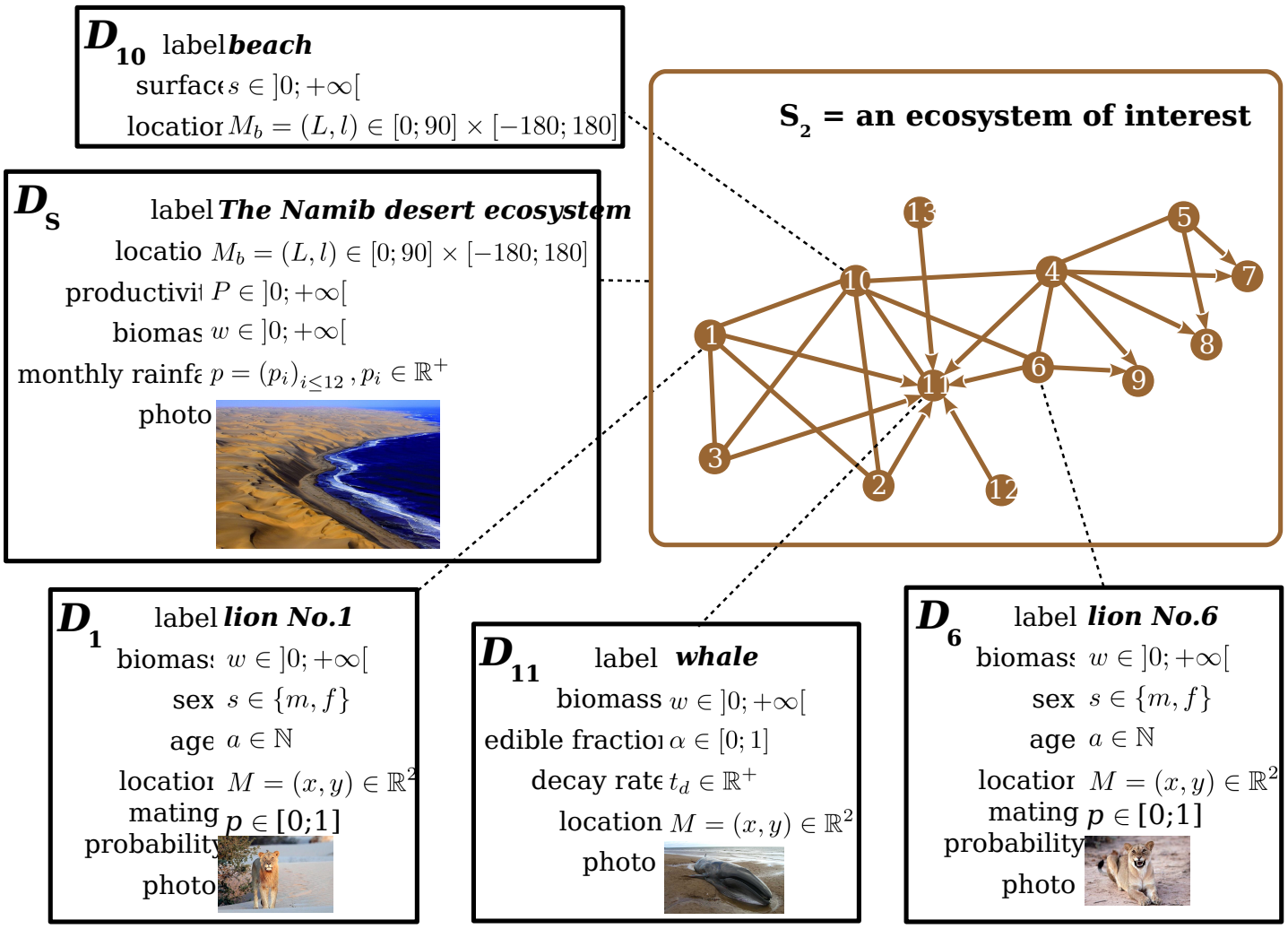


Figure 2. An example of graph descriptors (redrawn from Gignoux et al. 2017).

1.2.3. Categories

In an IBM, every individual differs from all others. But it is common practice to assume that some groups of individuals share some things in common. This is the essence of modelling: finding commonalities within an ocean of particular cases. We use the concept of *category* to group system components that 'look like each other' in some way. Categories are used to express how alike and how different certain groups of components are from each other.

In a dynamic graph, categories are used to specify common descriptors to groups of components (e.g. a plant species average growth rate, an animal cohort survival rate, ...) and functions that operate on a group of components of the same category. They are also used to specify which *type of relation* is possible between components of different categories.

This category concept is similar in many ways to the class concept used in the UML (http://uml.org/what-is-uml.htm) language and in object oriented programming (https://en.wikipedia.org/wiki/Object-oriented_programming). Categories can be nested, and can be exclusive of each other or not, that is, something can belong to one category of a set of caterories but not others within the same set e.g you can be *ephemeral* or *persistant* but not both. They are central to the organisation and execution of a simulation in **3Worlds**.

1.2.4. Time with simultaneous events

Because ecosystems have both biological and physical characteristics, they not only deal with individuals and populations of living organisms, but also with fluxes of matter and energy. To properly compute a mass or energy balance typical of physical questions, we need a time model that insures that all system components are modified synchronously - otherwise, leaks in mass and energy budgets may occur (Figure 3). This is where IBMs differ somewhat from MASs in their most common current implementations: MASs emphasize the *autonomy* of agents by allowing them to modify their state immediately. In other words, MASs assume that no two events occurring in a simulation can be simultaneous, while mass/energy balance requires simultaneity of events. 3Worlds assumes that simultaneous events are the default, but by using particular time models it is possible to relax this constraint.

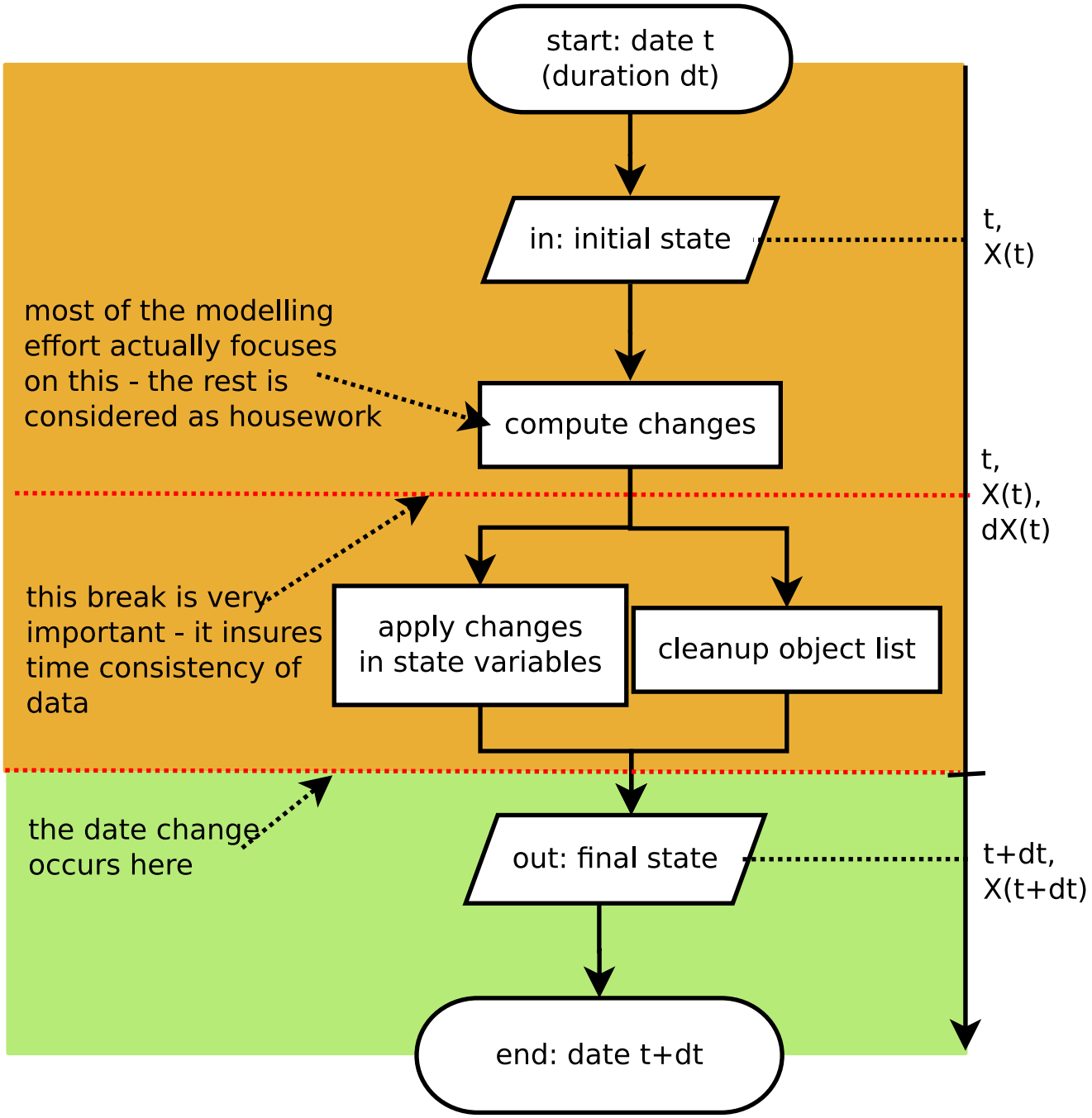


Figure 3. The time model of 3Worlds. The overall dynamics of a system is computed as X(t+dt) = f(X(t)) where X represents the state of all system components, t is time and f is the function (or rather, the set of functions) used to compute the change over the time interval dt.

Given the diversity of topics subject to ecological modelling, ecological processes span many orders of magnitude in their rate of action. Demographic models often use a yearly time step, while ecophysiological models may use daily time steps and physiological models may run with time steps of a second. in **3Worlds**, time steps span the whole scale

of time units relevant to ecology, i.e. from milliseconds to millenia. Standard Gregorian calendar time can also be used. **3Worlds** provides three types of *time model* that can be used in interaction: '*clock*' models using a constant time step, *event-driven* time models where time events trigger computations that can generate further events in the future, and *scenarios* where the list of events determines when computations are to be made.

1.2.5. Space as a mediator for interactions

IBMs are often 'spatialized', which means they include some representation of the physical space in which organisms of an ecosystem interact. We argued in Gignoux et al. (2011) that space is not a requirement of the ecosystem definition, but rather an optional feature. Besides the usual 3-dimensional space, we called the place where organisms interact in an ecosystem the *arena*, i.e. the place where things happen and where a public (of ecological modellers) is watching them. This does not mean an Euclidian (or any other kind of) space must be associated with the ecosystem representation. Using a dynamic graph is sufficient.

However, it is often the case that explicitly considering space in a model is necessary for computing ecological interactions. In most ecological process models, there are actually implicit assumptions about space and how it affects organism interactions. For example, seed dispersal in plants is easily computed in a 2-dimensional space where the location of seeds depends on that of parent plants and some simple distance law; water flow in a catchment relies on a 2-dimensional space plus an elevation of ground surface to some x and y resolution; competition between individual trees in a forest assume a vertical distribution of leaves depicted with varying degrees of detail.

Following Gignoux et al. (2011), we optionally provide predefined spatial representations to include in a simulator. Different spaces can be used within a single simulator, depending on the needs of the process computations. They are associated with optimal search algorithms (e.g. Kd-trees) that speed up the search of components with which to form dynamic relations.

1.2.6. Modelling made easy

The community of ecological scientists has been developing an impressively large number of models, yet most of them are poorly designed in terms of programming, as ecologists are not necessarily software engineers. Ecosystem simulators are among the most complex programs (Coquillard & Hill 1997). They require high programming skills and constitute a huge investment in time, which makes their production slow and hazardous. As a result, once built, they tend to be used beyond their initial domain of application (e.g. the overuse and abuse of the CENTURY model: Parton et al. 1988), issues of provenance and repeatability are rarely addressed, shedding some doubt on the discipline as a whole.

With **3Worlds**, we wanted to provide a simulation platform for ecosystem modelling using state-of-the-art concepts and algorithms, and sound programming techniques (e.g. systematic code testing, separated concerns), so that ecological modellers can concentrate on the ecological part of the problem and forget about the computer science part. We used *automatic code generation* to ensure that researchers need only edit one code file to build a simulator for their particular model. We used a *graph editor* to build the configuration and organise the data required for a particular study. In **3Worlds** therefore, an ecosystem model only requires two files: a specification file organised as a graph, and a computer code file where all relevant ecological processes are written. None of this prevents the modeller from using software libraries, either their own or from a third party, to extend their coding capabilities.

When designing a model, it is important to get quick visual feedback system behaviour when one changes equations or their implementation. **3Worlds** comes with a library of user-interface objects (graphs, maps, time series) that can be freely assembled to adapt outputs to the needs of the researcher.

1.2.7. Model comparison: graphs can be compared

Climate change modelling relies on 19 major general circulation models (GCMs) all based on the same equations. When run with identical datasets (initial data plus forcings), they all yield different results. This is expected given the size of their code, but what is troubling is that nobody is able to trace within the code where the differences come from (Lim &

Roderick 2009). This problem arises again and again in the modelling literature (e.g. Melilo et al. 1995; Roxburgh et al. 2004). The ultimate reason for this impossibility is that all the knowledge invested into these huge models is represented in computer code, which are very difficult to compare for any but the simplest of models.

3Worlds is an attempt to solve this issue *for the future* (there is nothing we can do for past model code). If models are developed within the standard framework of **3Worlds**, the only thing that needs to be compared among models is their specification file (a graph) and their code file - hundreds to thousands of lines, not more. Everything else is equal. In theory this should facilitate model comparison.

1.2.8. Problem upscaling

Developing a simulator is only a small part of the ecological modelling exercise: once the simulator is ready, it is used as a real ecosystem in *simulation experiments*. Designing and running such experiments is a very important part of the job - if not the most important, as it is the one which gives insight and publishable results.

IBMs are often stochastic, as population rates translate into probabilities at the individual level: e.g., the code has to decide which individuals to delete to satisfy a mortality rate of 10%. This is usually based on drawing random numbers. As a result, every simulation is different even when using identical parameters, and an asymptotic behaviour of the system can only be obtained by running multiple simulations. Fortunately, this is easily parallelized with modern computers.

3Worlds is interfaced with OpenMole (https://openmole.org/) to provide access to big computing power. Through OpenMole, big simulation experiments can be deployed on networks of computers, grids, or supercomputers.

1.2.9. Portability and performance

3Worlds is written in Java to ensure portability between all operating systems. Its code has been carefully optimised, although generality inevitably comes with some performance cost.

Cited references:

Bousquet, F., & Le Page, C. (2004). Multi-agent simulations and ecosystem management: a review. *Ecological Modelling*, 176:313–332. https://doi.org/10.1016/j.ecolmodel.2004.01.011

Coquillard, P., & Hill, D. (1997). *Modélisation et simulation d'écosystèmes. Des modèles déterministes aux simulations à événements discrets*. Masson, Paris.

Flint, S. R. (2006). *Aspect-Oriented Thinking - An approach to bridging the disciplinary divides*. PhD, Australian National University.

Gignoux, J., I.D. Davies, S.R. Flint, & J.D. Zucker (2011). The Ecosystem in Practice: Interest and Problems of an Old Definition for Constructing Ecological Models. *Ecosystems* 14: 1039-54. https://doi.org/10.1007/s10021-011-9466-2.

Gignoux, J., G. Chérel, I.D. Davies, S.R. Flint, & E. Lateltin (2017). Emergence and Complex Systems: The Contribution of Dynamic Graph Theory. *Ecological Complexity* 31: 34-49. https://doi.org/10.1016/j.ecocom.2017.02.006.

Grimm, V., & Railsback, S. (2005). *Individual-based modelling and ecology*. Princeton University Press.

Harary, F., & Gupta, G. (1997). Dynamic graph models. *Mathematical and Computer Modelling*, 25(7), 79–87. https://doi.org/10.1016/S0895-7177(97)00050-2

Lim, W. H., & Roderick, M. L. (2009). *An atlas of the global water cycle based on the IPCC AR4 climate models.* ANU E Press.

Melilo, J. M., Borchers, J., Chaney, J., Fisher, H., Fox, S., Haxeltine, A., Janetos, A., Kicklighter, D. C., Kittel, T. G. F., McGuire,

A. D., McKeown, R., Neilson, R., Nemani, R., Ojima, D. S., Painter, T., Pan, Y., Parton, W. J., Pierce, L., Pitelka, L., … Woodward, F. I. (1995). Vegetation/ecosystem modeling and analysis project: comparing biogeography and biogeochemistry models in a continental-scale study of terrestrial ecosystem responses to climate change and CO_2 doubling. *Global Biogeochemical Cycles*, 9(4), 407–437.

Parton, W., Stewart, J., & Cole, C. (1988). Dynamics of C,N, P and S in grassland soils: a model. *Biogeochemistry*, 5, 109–131.

Roxburgh, S. H., Barrett, D. J., Berry, S. L., Carter, J. O., Davies, I. D., Gifford, R. M., Kirschbaum, M. U. E., McBeth, B. P., Noble, I. R., Parton, W. G., Raupach, M. R., & Roderick, M. L. (2004). A critical overview of model estimates of net primary productivity for the Australian continent. *Functional Plant Biology*, 31(11), 1043–1059.

Tansley, A G. (1935). The use and abuse of vegetational concepts and terms. *Ecology* 16: 284-307.

Zucker, J.D. (2003). A Grounded Theory of Abstraction in Artificial Intelligence. *Philosophical Transactions of the Royal Society B: Biological Sciences* 358: 1293-1309. https://doi.org/10.1098/rstb.2003.1308.

2. Getting started - download and installation

2.1. Basics - what you must know before starting

3Worlds is an application designed to develop and launch simulations of ecosystems. It is highly versatile and can simulate any kind of ecosystem using any kind of mathematical logic.

The application and use of 3Worlds to a particular ecosystem for a particular study case is called a *model* — or, more precisely a *simulation* model. The model must first be specified and developed (this involves writing some code in the java programming language) before it can be executed for a particular case study. This execution is called a *simulation experiment*.

3Worlds comprises two main applications:

- `ModelMaker`, to configure a model;
- `ModelRunner`, to run the model.

Creating a model involves creating a configuration with `ModelMaker` and developing some associated Java code to specify details particular to your model. To do this, you must use the Eclipse (https://www.eclipse.org/downloads/) programming software (freeware). Later versions of 3Worlds may support other packages, but at the time of writing, 3Worlds will only work with eclipse (https://www.eclipse.org/downloads/).

`ModelMaker` will generate Java code for data structures specific to a model (based on the configuration file you have developed) and *template* java code for each process you have defined. These process templates are where you enter programming code to implement your model. You only need to write code for your processes and for model initialisation. All else is managed by 3Worlds.

3Worlds is written in java (https://en.wikipedia.org/wiki/Java_(programming_language)), which makes it OS-independent. It can be run on MacOS, Linux or Windows computers.

2.2. Prerequisites

You will need the following software installed on your computer before installing 3Worlds:

- Java JDK (**J**ava **D**evelopment **K**it), version 11 or greater (Oracle (https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html) or open (http://openjdk.java.net/)

version)

- Java fx (graphical user interface library for java: Oracle (http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html) or open (http://openjdk.java.net/projects/openjfx/) version)
- an Eclipse (https://www.eclipse.org/downloads/) Java development environment and add the e(fx)clipse plugin (https://www.eclipse.org/efxclipse/install.html) required for Javafx

2.3. Running ModelMaker

This assumes you have downloaded `3w.zip`.

1. Unzip `3w.zip` in your user home directory (important), keeping the internal directory tree. This will extract a file `modelMaker.jar` and a `.3w` directory containing more jar files.
2. Double-click on `modelMaker.jar`. This should launch the ModelMaker application.
3. If this doesn't work, open a terminal and type `java -jar modelMaker.jar`. This should launch the ModelMaker application.

This way of launching 3Worlds is sensitive to the relative location of `modelMaker.jar` and the other jars that are in the `.3w` directory, ie `modelMaker.jar` **must** be exactly one directory above `.3w`.

To remove this constraint, you can bypass `modelMaker.jar` by directly launching it from the jars contained in `.3w`:

```bash
java -cp .3w/tw-dep.jar au.edu.anu.twuifx.mm.MMmain
```

It is preferable to run the software from a terminal window. This way, any errors that may arise will be displayed.

to develop your model-specific code, you will need to setup a Java development environment as shown in Section 2.4.

2.4. Setting up a Java development environment for the user code

2.4.1. Setting up the Eclipse Integrated Development Environment (IDE) for 3Worlds

This assumes you have downloaded `tw-dep.jar`.

1. If not yet done, install Eclipse (https://www.eclipse.org/downloads/) (don't forget e(fx)clipse (https://www.eclipse.org/efxclipse/install.html)!)
2. Create at *workspace* (a working directory for Eclipse - Eclipse will ask for it when launched). e.g., <*my_workspace*>
3. Within Eclipse, create a *project*:
   - Select menu `File → New → Java project`; this opens a dialog box
   - In the dialog box, type a project name (e.g. <*my_project*>)
   - Click the `Finish` button
4. Import 3Worlds dependencies:
   - Select menu `Project → Properties`; this opens a dialog box
   - In the dialog box, select `Java Build Path`
   - Select the `Libraries` tab

- Select `ClassPath`
- Click on the `Add external JARs…` button; this open a file selection dialog box
- In the file selection dialog box, browse and select `tw-dep.jar`
- Click the `Apply and Close` button

2.4.2. Running ModelMaker from Eclipse

ModelMaker can be run from Eclipse or as a standalone application since it is included in the `tw-dep.jar` library required to develop the user code.

- In the `package explorer` window, expand the `Referenced libraries` entry
- Right-click on the `tw-dep.jar` entry, select `Run as → Java Application`. This opens a dialog box
- In the dialog box, type *Main*
- In the list of matching items, select `Main - au.edu.anu.twuifx.MMmain` and click `OK`
- If warning errors appear, click `Proceed`. This launches the ModelMaker application

2.4.3. Linking user code with model configuration

This requires the following actions:

1. In ModelMaker,
   - create or open a *3Worlds* project (`Projects` entry of the main menu)
   - select `Preferences → Java Project → Connect…`. This opens a dialog box with a file selector
   - select the root directory of the *Eclipse* project as created above (e.g. *<my_workspace>/<my_project>*)

   This operation tells `ModelMaker` to generate its code into the user java project. When you want to edit your code in eclipse, you must first **refresh** the eclipse project:
2. In Eclipse,
   - select the project name at the very top of the `package explorer` window
   - right-click on it and select `Refresh`
   - or, alternatively: press the **F5** key

You don't **have** to do this. We provide it as a facility if you want to run ModelMaker from eclipse rather than directly for some reason of your own.

2.4.4. Debugging and testing user code

The user code, first generated by `ModelMaker` and further edited by the user, can be run using `UserCodeRunner.java`. This class can be found in the `default` src directory and was created when linking this project with the 3Worlds project (cf above). It requires three command line arguments (we assume that you know how to setup and run a `Run Configuration` in Eclipse):

- an instance number (more about this later); leave this at '0' for now.
- the name of the directory of the 3Worlds project as created by `ModelMaker` (e.g. *project_test_model9_2019-09-05-08-50-20-458*). This project directory is located under the `.3w` directory automatically created by `ModelMaker` as its working directory
- some optional settings to switch on debugging logs.

With this, the user code should be executed as a test simulation by `UserCodeRunner`.

Further edits and modifications of the configuration can be made in `ModelMaker`, but don't forget to keep the Eclipse project content synchronized with the ModelMaker project by refreshing the Eclipse project as often as necessary.

3. Configuration reference: creating and editing a model with ModelMaker

3.1. General concepts: structure of a 3Worlds configuration

3.1.1. A tree structure...

The configuration of a 3Worlds *simulation experiment* is organised as a tree (Figure 4). Each tree *node* specifies a subset of the parameters of the whole configuration. Each *node* has *child nodes* linked through a *tree edge*, so that large pieces of configuration can be broken down into the relevant details. At each level of this hierarchy, *properties* can be attached to *nodes*.

Nodes have a *label* and a *name* that are displayed in the `ModelMaker` interface as `label:name`:

- The *label* specifies what role this particular *node* plays in the whole configuration. For example, the *node* labelled `experiment` is used to configure a *simulation experiment*.
- The *name* is used to differentiate *nodes*. It must be unique over the whole configuration tree. By default, `ModelMaker` generates unique names (by adding a number to the name if replicates are found).

The *root node* of a configuration is always labelled `3worlds`.

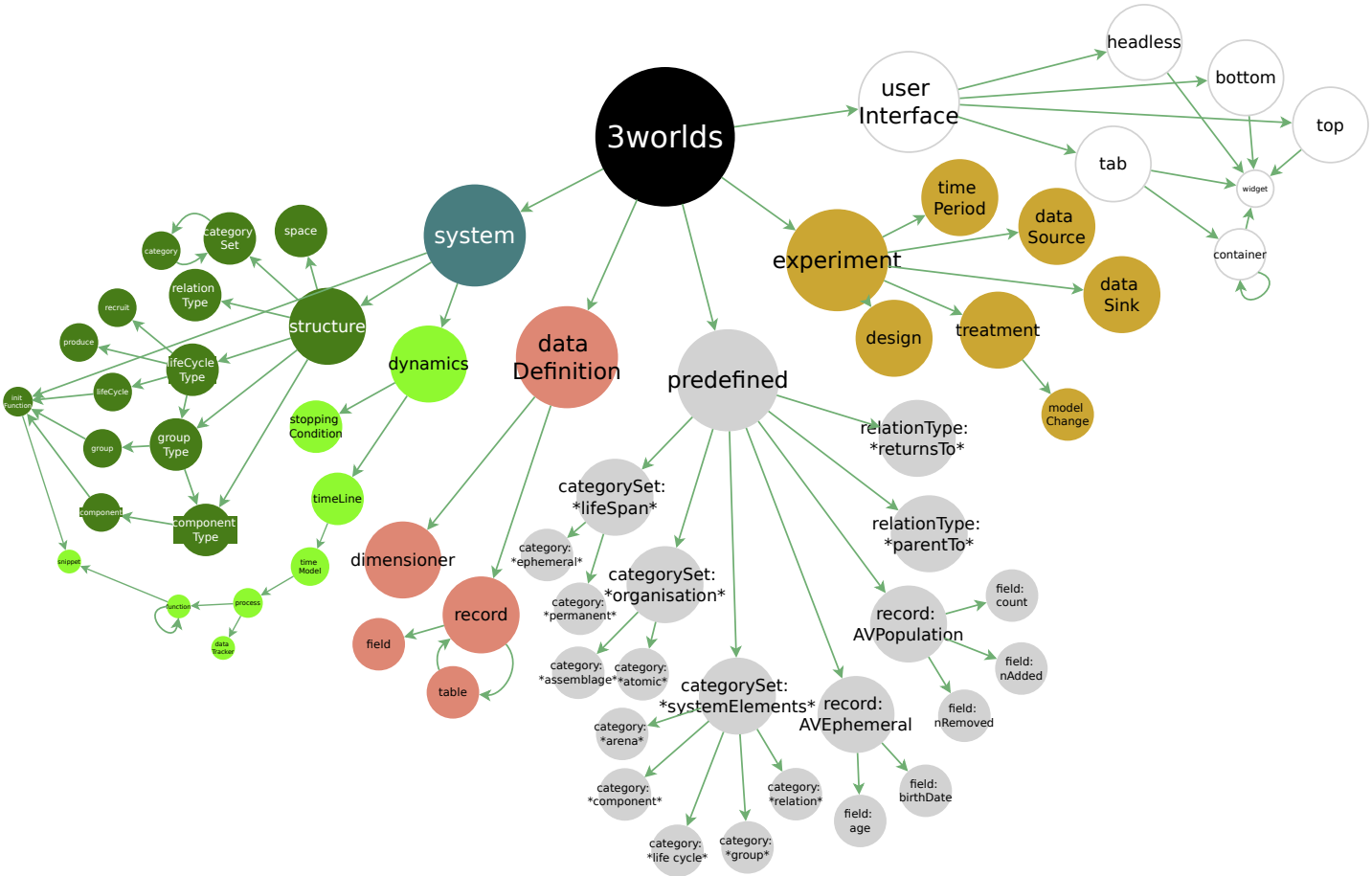


Figure 4. Tree structure of a simulation experiment configuration in 3Worlds

The configuration tree is stored in a file in a specially designed text format, ending with the extension `.ugt` or `.twg`.

Such files are produced by `ModelMaker` and can then be exchanged and imported into `ModelMaker` via the `Projects → Import…` menu entry. Their format is human-readable, but they must **never** be edited with another software than `ModelMaker` - **the risk is to corrupt all your configuration** and be unable to run it (or even edit it with `ModelMaker` again).

NEVER edit the 3worlds configuration `.ugt` file 'by hand'! You may make it unreadable to `ModelMaker` and lose all your work.

Each node in the configuration has a particular meaning for `ModelRunner` : the configuration must comply with certain rules and constraints, the first one being the particular set of *nodes* that have been designed and appear on Figure 4. The detailed meaning of all *nodes* and their *properties* is described in Section 3.3.

3.1.2. ...with cross-links

Actually, the 3Worlds configuration is not strictly hierarchical (Figure 5): according to their role in `ModelRunner` , some configuration *nodes* need to gather information from other parts of the configuration tree. This is done by allowing for some cross-reference *edges* to be defined, that overlay with the strict hierarchical structure of the tree. As a result of these cross links, the whole configuration is a *graph* rather than a tree.

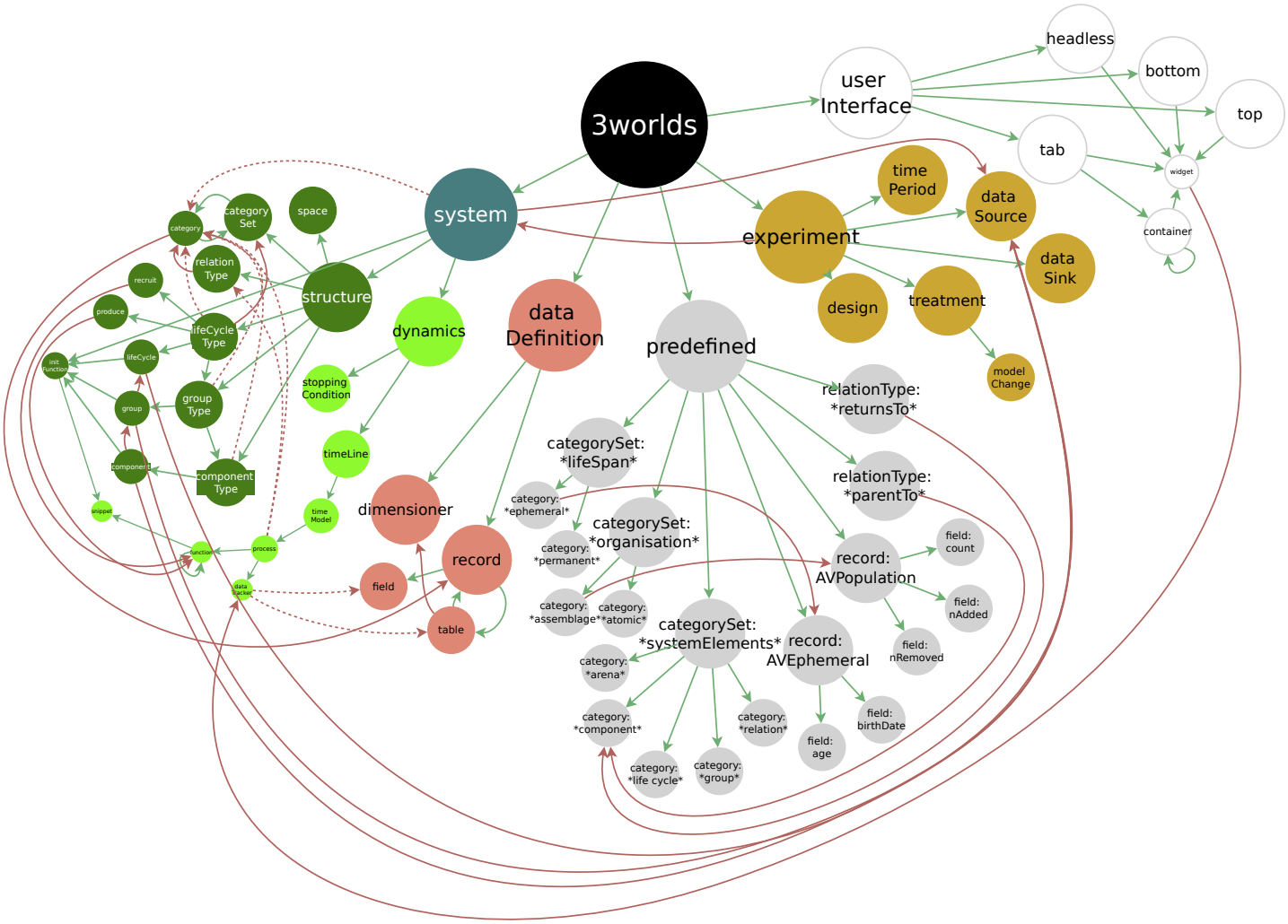


Figure 5. Tree structure of a simulation experiment configuration in 3Worlds, showing the cross-links between tree nodes (NB: not all cross-links are drawn)

Edges representing cross links have a *label*, *name*, and may have *properties* just as *nodes* do. The detailed meaning of all cross-reference *edges* and their *properties* is described in Section 3.3.

3.1.3. What ModelMaker does for you

`ModelMaker` knows the details of the configuration constraints. It facilitates the design of a configuration by only letting you add the *nodes*, *edges* and *properties* that will produce a valid, runnable configuration file. During the configuration building process, it constantly checks the validity of the graph and reports any errors or missing parts in its `log` panel. `ModelMaker` is far more than a nice visual editor producing a graph: a configuration graph produced with `ModelMaker` is **guaranteed** to run with `ModelRunner` because of all these internal consistency and validity checks.

3.2. Using ModelMaker: software interface and functioning

TO DO: step-by-step description of using the user interface. With screenshots.

3.3. Configuration options: reference

In this section,

- node and edge labels are indicated in `bold`
- text in triangular brackets (`<>`) mean a user-defined value is expected; the text usually specifies what kind of value is expected (e.g. `<name>` for a name, `<int>` for an integer number, etc.). If the text is required, it will be underlined, otherwise it is optional
- a *multiplicity* in curly braces {} tells how many times the item may appear in a configuration:

  {1} exactly one item is required

  {0..1} the item is optional, i.e. one or zero is required

  {1..*} one to many items are required

  {0..*} any number of items is possible

- levels in the tree hierarchy are indicated by slashes `/` .
- a column `:` separates a node label from its name.

3.3.1. The *3Worlds* node

`/3worlds:<name>` {1}

This node is the root of any 3Worlds configuration file (Figure 6). The name will appear in `ModelMaker` 's main window title, in the **project directory name** and in the **configuration graph file**. The name is requested and set when creating a new project (`Projects>New` menu entry in `ModelMaker`).

Properties for `3worlds`

`authors`	The list of this project author names.
`contacts`	The list of the author contacts (e.g. e-mail addresses), in the same order as the author names (1-1 match).
`precis`	A short description of the model contained in this project.
`publication`	A list of bibliographic references in relation with the project.
`built-by`	The user name under which this project was constructed and saved, together with the date of saving. This property is automatically set by ModelMaker.

`version` A user-defined version identifier for this project.

All these properties appear in the about box of `ModelRunner` and are saved in the ODD file describing the model.

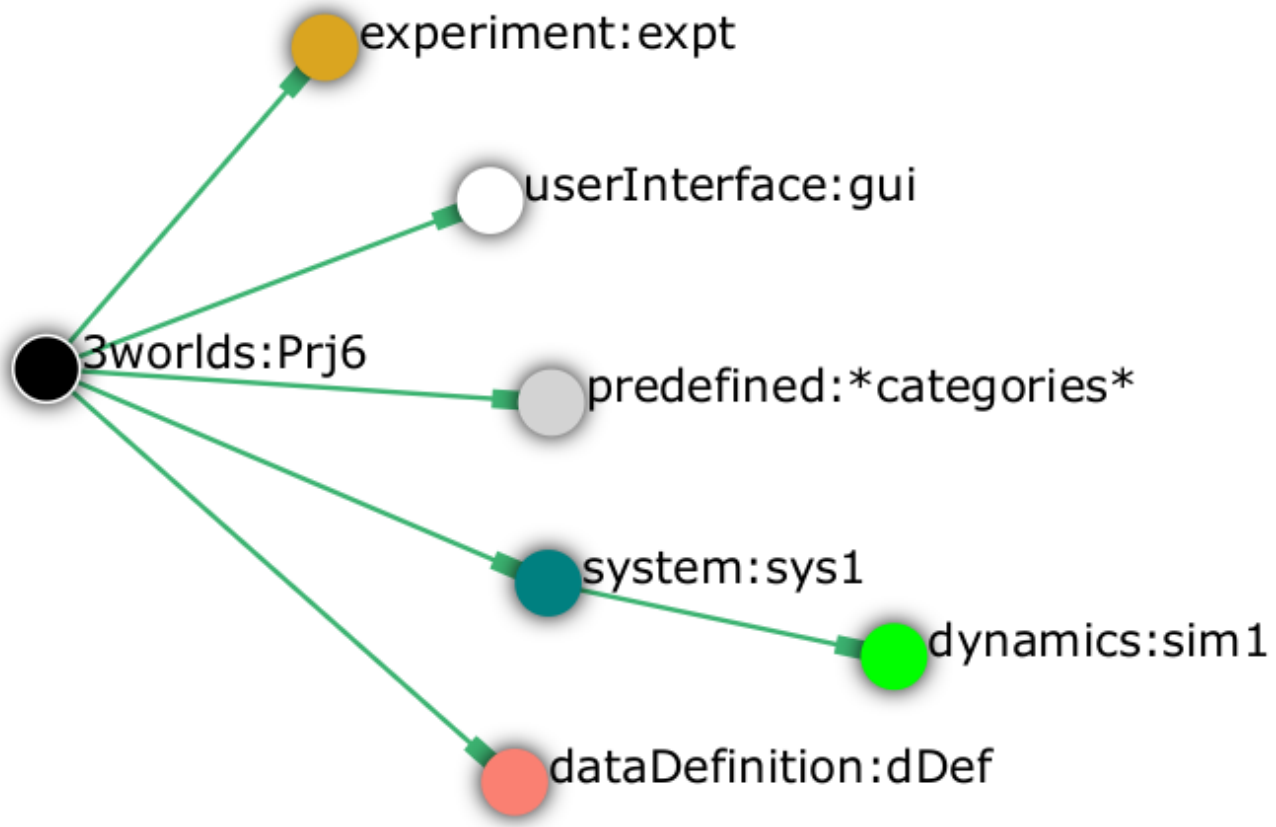


Figure 6. The base tree of any 3Worlds configuration

3.3.2. The *system* node

`/3worlds/system:<name>` {1..*}

This node and its sub-tree contains all the ecological concepts used to define a simulation model: what entities are modelled, what biological processes apply to them, at what time step they should run. The name is used to differentiate models as a simulation experiment may involve more than one model.

A 3Worlds ecosystem is constructed as a population of 'biological entities', called *system components*, which establish interactions called *system relations*. Components and relations have *variables* that may change value over time, according to various *ecological processes* attached to components. In other words, an ecosystem is represented as a *dynamic graph* which changes state and structure over time. To reflect this, `system` has two sub-trees called `structure` and `dynamics`.

For very simple models, where the modelled system is not further subdivided into components, the structure sub-tree is not required. In this case,the system node will be used as a unique 'component' describing the system (cf tutorial: logistic).

Cross-links for `system`:

`belongsTo → category:<name>` **{1..*}**

This link tells to which categories the system node belongs. The categories must not belong to the same category set.

If there are nested categories, membership is inherited.The categories targets of these links must be specific to the system, i.e. no `componentType`, `groupType` or `lifeCycleType` should refer (`belongTo` cross-link) to any of them.

`belongsTo → category:*arena*` **{1}**

This link to the predefined category `*arena*` is required in all models (Section 3.3.8). The arena is 'the place where things happen', i.e. it describes the part of the system where components interact. If no components are modelled, then it is the system.

`belongsTo → category:*atomic*|*assemblage*` **{1}**

This link to either the predefined category `*atomic*` or `*assemblage*` is required in all models. Only if the system is very simple (no components, no groups, no life cycles) should `*atomic*` be selected. It means the system has no further subdivision. In any other case (i.e. as long as there is a structure sub-tree), the system should be an `*assemblage*`. Assemblages have three automatic variables: the system size (number of components) `count`, the number of new components added during last time step `nAdded`, and the number of components deleted during the last time step `nRemoved`.

`belongsTo → category:*permanent*` **{1}**

This link to the predefined category `*permanent*` is required in all models. It specifies that the system is going to stay forever (permanent) during a simulation.

3.3.3. The *system/structure* node

`/3worlds/system/structure:<name>` {0..1}

This node and its sub-tree contains the description of the system component and relation types. It is based on the concept of *category*. A category is a set of components with an identical description, e.g. the same set of state variables or the same growth function.

The *category* and related concepts: specification of groups of entities

3Worlds uses these concepts to specify and generate the ecological entities manipulated during a simulations.

Category

`…/categorySet/category:<name>` {1..*}

A `category` is simply a name attached to a set of objects sharing common properties (Section 1.2.3). These common properties are descriptors (Section 1.2.2), and dynamic behaviours (or *processes*). Categories, grouped into `categorySet`s, constitute a user-defined *classification* of *system component types* relevant for a particular model.

Descriptors consist in *drivers*, *decorators*, *constants* and *automatic variables*:

drivers

are *independent* variables (numbers, text, logical values) that characterize the state of a system component at an instant in time (*e.g.* biomass, age, sex, social status…). They will vary during a simulation. 'Independent' means that these variables cannot be computed from each other within a simulation step, they also depend on their previous step values. For example, if some category is described by drivers *age* and *biomass*, neither of them can be computed from the other: *age* is computed from its previous value by adding the time step duration; *biomass* is computed in some other way probably involving its previous value. The values of drivers are carried over from a simulation step to the next, i.e. they *drive* the dynamics of the system.

decorators

are *dependent* variables that are computed from drivers and other decorators within a simulation step. For example, *leaf area* could be computed as some constant times *biomass*: its value is completely determined by the value of

biomass. As such, decorators are not independent from the drivers and are used only for convenience in computations or output display. They are not carried over from a simulation step to the next, they are always automatically reset to zero at the end of each step.

constants

are values (numbers, text, logical values) that do not change during a simulation. For example, *sex* or *number of legs* are not going to change during the lifetime of animal. They are set at birth and never changed after.

automatic variables

are values internally managed by 3Worlds and only available as read-only values for computation. For components representing individuals, these are *age* and *birth date*. For components representing populations, these are *number of individualls*, *number of births*, *number of deaths*.

The exact data structures for descriptors are specified under the `dataDefinition` node (Section 3.3.5) and linked to the category through the following:

Cross-links for `category`:

`drivers → record:<name>` **{0..1}**

`decorators → record:<name>` **{0..1}**

`constants → record:<name>` **{0..1}**

These links tell which data structure in the `dataDefinition` node (Section 3.3.5) is used to store *drivers*, *decorators* and *constants*.

Automatic variables do not need to be specified - because they are automatic.

A category may be defined with no drivers, decorators or constants.

There are *predefined categories* in 3Worlds under the node `/3worlds/predefined` (Section 3.3.8), used to specify particular components of the system. They will be explained in time.

CategorySet

`/3worlds/system/structure/categorySet:<name>` {1..*}

Some categories must be exclusive of each other: for example, an ecological entity is either a plant or an animal, but can't be both. For this reason, *exclusive* categories are grouped into `categorySets`. A `categorySet` is *a set of* ***mutually exclusive*** *categories*. Categories can be nested, by simply defining a `categorySet` as a child of a `category`:

`…/category/categorySet:<name>` {0..*}

RelationType

`/3worlds/system/structure/relationType:<name>` {0..*}

A `relationType` is just a name representing a meaningful link between two kinds of categories. It is specified by giving it a name and cross-linking it to the relevant categories with `fromCategory` and `toCategory` cross-links. Note that a `relationType` can link more than one 'from' categories to more than one 'to' categories if required. `RelationType`s are used to implement specific processes acting on ecological entities (for example, a predation process).

Cross-links for `relationType`:

`fromCategory → category:<name>` **{1..*}**

This link tells which categories are at the start of the relation.

`toCategory → category:<name>` **{1..*}**

This link tells which categories are at the end of the relation.

Properties for `relationType`:

`lifespan` This property specifies if this type of relation will stay attached to its `systemComponents` during all their life, or may get created and deleted during their lifespan.

> *possible values*:
>
> `permanent` relation stays as long as both its ends stay (default value)
>
> `ephemeral` relations are created during a simulation by the means of a `relateTo` function, and kept or deleted by means of a `maintainRelation` decision function. If no `maintainRelation` function is provided, then the relation only lasts for one simulation step.

Example: a category tree

On this diagram (generated with `ModelMaker`), hierarchical links are in green and cross-links are in red.

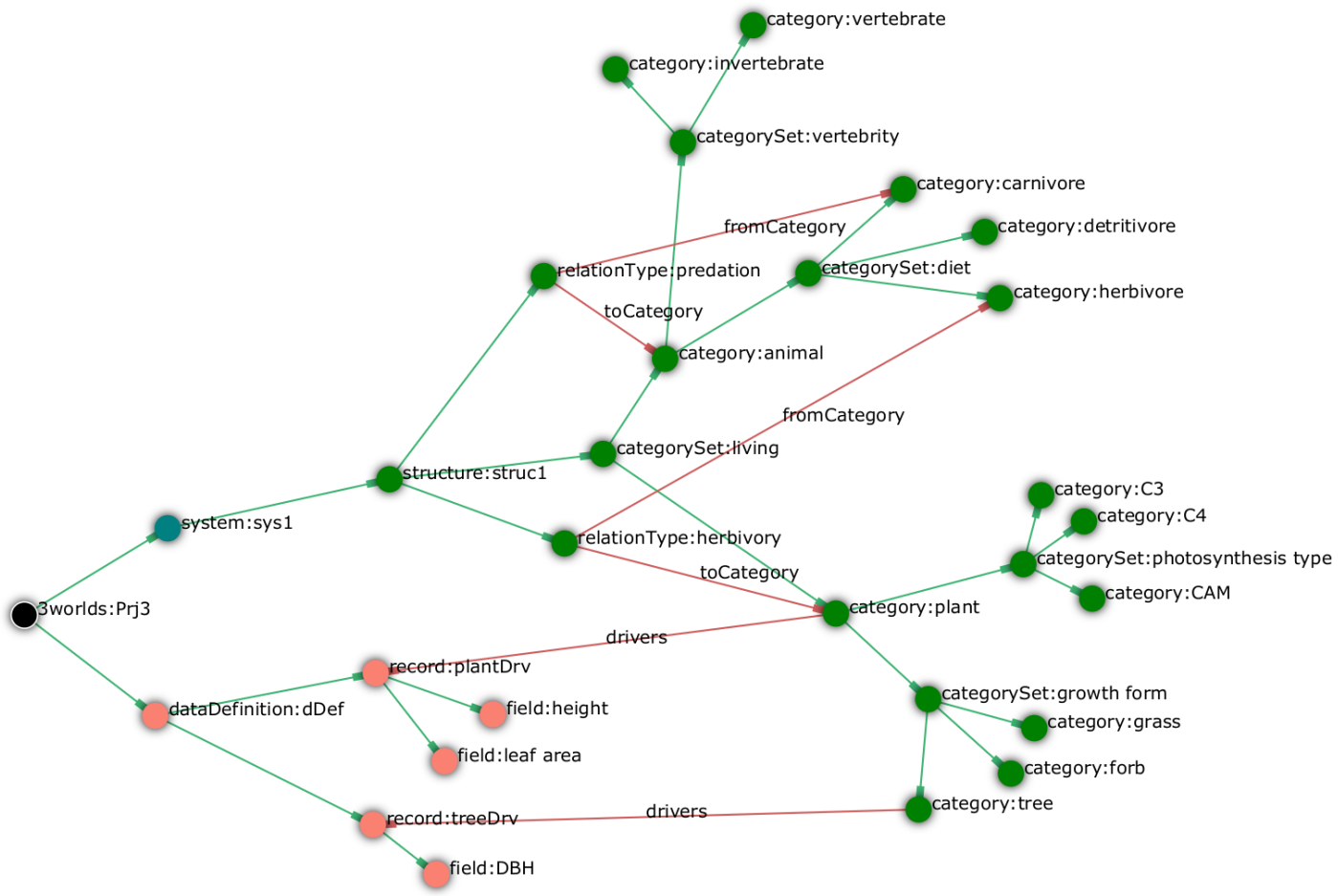


Figure 7. Example of a configuration with category sets, categories and relations

In this example, a *plant* can be a *C3 tree* but cannot be simultaneously a *grass* and a *forb*. Similarly, an *animal* cannot be both *herbivore* and *carnivore*. The *predation* relation links an *animal* of any kind (the prey) to a *carnivore* (its predator). A member of category *plant* will have two driver variables, *height* and *leaf area*. A member of category *tree* will have three driver variables, *height*, *leaf area* and *DBH*, as it inherits all the properties of the *plant* category in which it is nested.

The specification of ecological entities: *system components*

System component

`/3worlds/system/structure/componentType:<name>` {1..*}

3Worlds simulates a *system* made of *system components*. These are the things which are instantiated at run time, hold descriptors, and are dynamically changed over the time course of a simulation. When setting up a simulation, one must attach *categories* to *system components*. The rules prevailing to build up category hierarchies mean that a system can belong to a number of non-exclusive categories, as long as the exclusion and nesting rules are respected. For example, we could define a system as belonging to the *plant* and *tree* categories, but not to the *animal* and *tree* categories.

Cross-links for `componentType`:

`belongsTo → category:<name>` **{1..*}**

This link tells to which categories a system component type belongs. The categories must not belong to the same category set. If there are nested categories, membership is inherited (e.g. in the previous example, belonging to the *C3* category automatically implies the system component is also a *plant*).The categories targets of these links must be specific to component types, i.e. no `groupType` or `lifeCycleType` should refer (`belongTo` cross-link) to any of them.

`belongsTo → category:*component*` **{1}**

This link to the predefined category `*component*` is required in all models.

`belongsTo → category:*atomic*` **{1}**

This link to the predefined category `*atomic*` is required in all models. It means that a component has no further subdivisions.

`belongsTo → category:*ephemeral*|*permanent*` **{1}**

This link to either of the predefined category `*ephemeral*` or `*permanent*` is required in all models. It specifies if components of this `componentType` are going to stay forever (permanent) or can be created and deleted (ephemeral) during a simulation. Ephemeral components have two automatic variables, `birthDate` and `age`.

Component group

`/3worlds/system/structure/groupType:<name>` {0..*}

Sometimes it makes sense to group components of a given `componentType` into *groups*; the best example of this in ecology is the species. Animals of different species may be described by the same descriptors, but nevertheless they will only reproduce within their species. Typically, animals of a species would share some constants, like plumage coloration, average body size, main diet etc. The `groupType` node is meant to fullfil this use case.

To make a `componentType` belong to a `groupType`, you just need to declare it as a child of the `groupType`:

`…/groupType/componentType:<name>` {0..*}

Cross-links for `componentType`:

`belongsTo → category:<name>` **{1..*}**

This link tells to which categories a system component type belongs. The categories must not belong to the same category set. If there are nested categories, membership is inherited. The categories targets of these links must be specific to groups, i.e. no `componentType` or `lifeCycleType` should refer (`belongTo` cross-link) to any of them.

`belongsTo → category:*group*` **{1}**

This link to the predefined category `*group*` is required in all models.

`belongsTo → category:*assemblage*` **{1}**

This link to the predefined category `*assemblage*` is required in all models. It means that a group is a population of components. Assemblages have three automatic variables: the group size (number of components) `count`, the number of new components added during last time step `nAdded`, and the number of components deleted during the last time step `nRemoved`.

`belongsTo → category:*permanent*` **{1}**

This link to the predefined category `*permanent*` is required in all models. It means that groups will stay forever in the system, even if all components of this groups are gone.

Life cycle

`/3worlds/system/structure/lifeCycleType:<name>` {0..*}

As *system component*s are designed to represent — among other things — individual organisms, they are able to create other system components at runtime, or to transform themselves into a system component of another category assemblage. These abilities are captured in a `lifeCycle`, which describes the possible creations and transitions of system components of a given category set into another.

Since `components` belong to `categories`, different types of system components represented by different state variables, subject to different ecological processes, can coexist in a simulation. It may occur in a particular model that one wishes to represent a transition between, e.g. development stages: think for example of a caterpillar turning into a butterfly. There are chances that you don't want to describe the caterpillar with the same variables and behaviours as the adult butterfly. The operation of transforming a system component from a selection of categories to another is called *recruitment*. Computationally, it means that the simulator must keep track of the system component's identity and age in the first stage and carry these properties on to the new system component of the second stage, and call an appropriate function to transform state variables of the first stage into the new one.

Reproduction is the second process by which system components of a given group of categories may produce other system components belonging to possibly different categories.

A specification of a life cycle requires:

1. a `categorySet` in which the categories represent the various stages of the life cycle;
2. at least one `groupType` definition per `category` of the above `categorySet`;
3. `recruit` and `produce` nodes linking categories within the above `categorySet` to describe possible recruitment and reproduction transitions;
4. optionnally, `categories` can be linked to a `lifeCycle` if one wants to attach descriptors to the life cycle.

To define a `groupType` within the context of a `lifeCycleType`, simply make it a child node of the lifeCycleType:

`…/lifeCycleType/groupType:<name>` {1..*}

Cross-links for `lifeCycle`

`appliesTo → categorySet:<name>` **{1}**

This links indicates which categories define the stages of the life cycle. `recruit` or `produce` nodes can only link categories of this set.

`belongsTo → category:<name>` **{0..*}**

These links enable to attach descriptors to a `lifeCycle`, if needed by some ecological processes. The categories targets of these links must be specific to life cycles, i.e. no `componentType` or `groupType` should refer (`belongTo` cross-link) to any of them.

`belongsTo → category:*life cycle*` **{1}**

This link to the predefined category `*life cycle*` is required in all models.

`belongsTo → category:*assemblage*` **{1}**

This link to the predefined category `*assemblage*` is required in all models. It means that a life cycle is a population of components. Assemblages have three automatic variables: size (number of components) `count`, the number of new components added during last time step `nAdded`, and the number of components deleted during the last time step `nRemoved`.

`belongsTo → category:*permanent*` **{1}**

This link to the predefined category `*permanent*` is required in all models. It means that life cycles will stay forever in the system.

Recruitment

`/3worlds/system/dynamics/lifeCycle/recruit:<name>` {0..*}

This node specifies that two categories of the life cycle `categorySet` are linked by a *recruitment* process.

Cross-links for `recruit`:

`fromCategory → category:<name>` **{1}**

This link tells which system component type is getting changed by the recruitment.

`toCategory → category:<name>` **{1}**

This link tells which system component type is the result of the recruitment.

multiple targets are possible from the same category, i.e. a component of a category may recruit to different categories. E.g., a bee larva can recruit to a worker or a queen.

`effectedBy → process:<name>` **{1}**

This link tells which ecological process is used to compute the recruitment. This process must implement exactly one `ChangeCategoryDecision` function.

Reproduction

`/3worlds/system/dynamics/lifeCycle/produce:<name>` {0..*}

This node specifies that two categories of the life cycle `categorySet` are linked by a *reproduction* process.

Cross-links for `produce`:

`fromCategory → category:<name>` **{1}**

This link tells which system component type is producing new system components.

`toCategory → category:<name>` **{1}**

This link tells which system component type is the result of the reproduction.

multiple targets are possible from the same category, i.e. a component of a category may produce components of different categories. E.g., a tree can produce seedlings through sexual reproduction and root suckers through vegetative reproduction.

`effectedBy → process:<name>` **{1}**

This link tells which ecological process is used to compute the production of new system components. This process must implement exactly one `CreateOtherDecision` function.

Example of a life cycle specification

This life cycle:

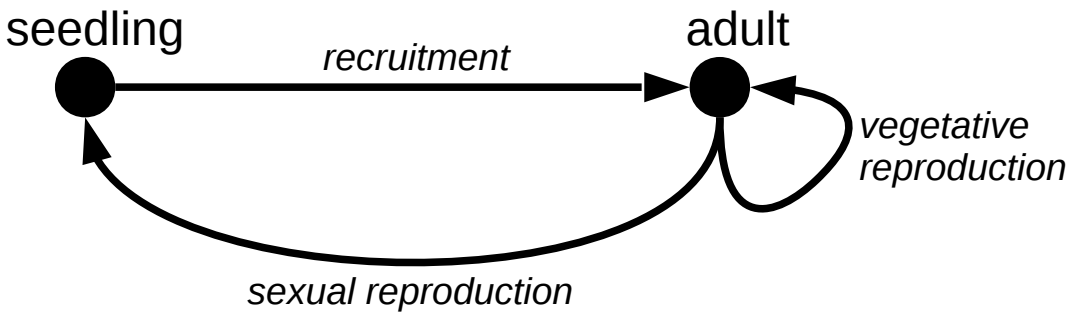


Figure 8. Example of a life cycle

is specified with this graph:

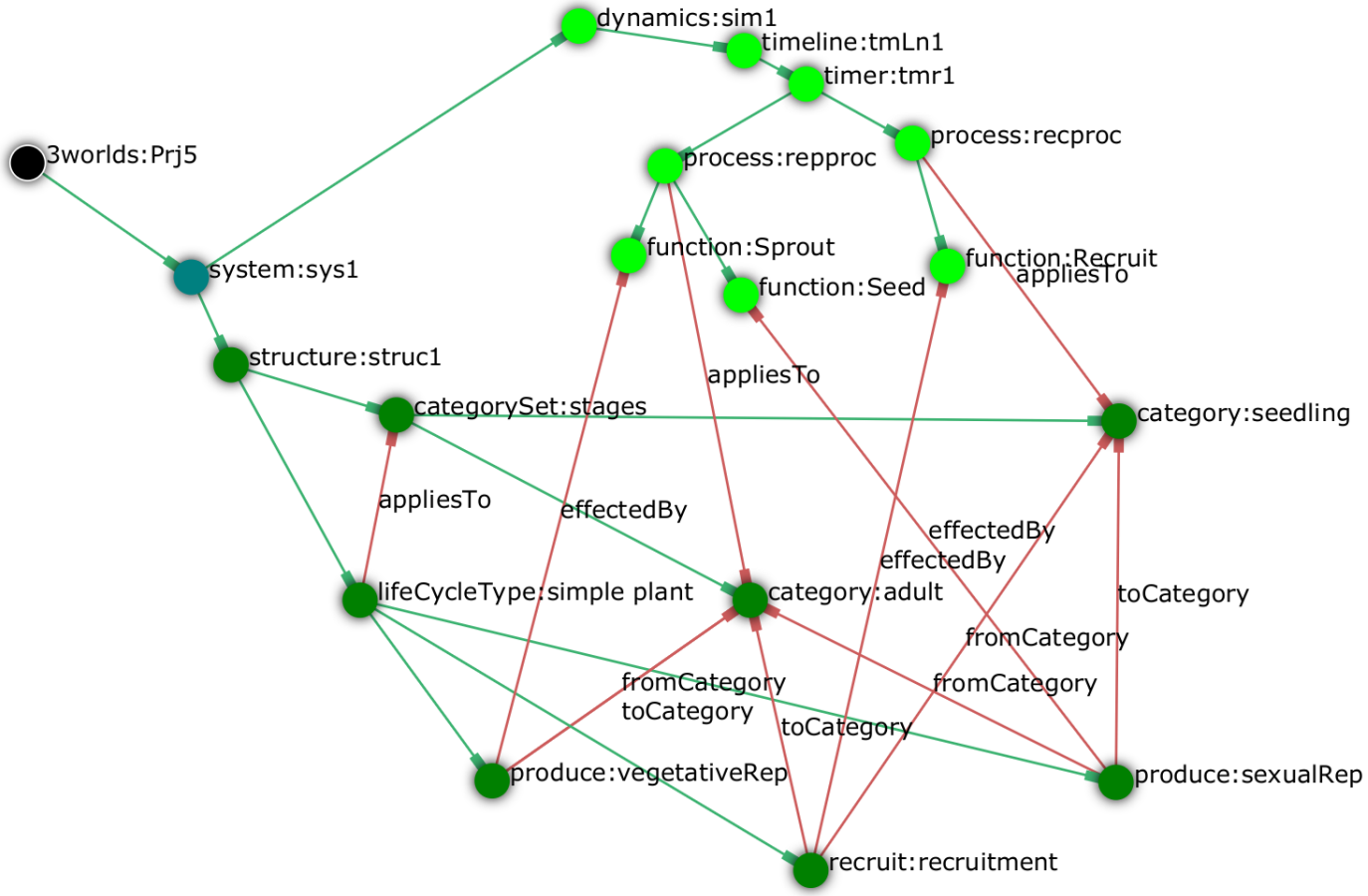


Figure 9. Example of a life cycle configuration

The life cycle has two categories, *seedling* and *adult*, and three transitions: a *recruit* transition called *recruitment*, effected by the function *Recruit*, and two *produce* transitions, effected by the functions *Seed* and *Sprout*.

The representation of space

Sometimes modellers want an explicit representation of space for their model. 3Worlds provides a library of different spaces for this need. Technically, space could just be treated as a component related to other components through particular relations. But since many optimal algorithms have been coded to handle particular aspects of space in relation to other entities, we use space as a kind of container for system components optimized for various operations such as searches for partners for establishing relations (see process).

Space and spatial indexing

`/3worlds/system/dynamics/space:<`*name*`>` {0..*}

This defines a space, i.e. a container where system components can be located by their *spatial coordinates* and where distances can be computed. More than one space can coexist in 3Worlds, applying to possibly different types of components and processes. Spaces can be of 1 to 3 dimensions (this is set by choosing a space type).

Properties for `space` :

`type` The type of spatial representation

possible values:

`continuousFlatSurface`	a flat plan with continuous coordinates (default value)
`squareGrid`	a flat square 2D grid for e.g. cellular automata
`topographicSurface`	a topographic plan with elevations **(not yet implemented)**
`linearNetwork`	a graph of connected segments for e.g., hydrologic networks **(not yet implemented)**

`borderType` The space border behaviour

possible values:

`wrap`	a soft teletransporting border which sends objects crossing it to the other side of the space (default value)
`reflection`	a hard border on which objects bounce back if they try to escape
`sticky`	a hard border on which objects stick if they try to escape
`oblivion`	a soft border into oblivion – objects that cross it disappear into nothingness
`infinite`	no border – space extends following object movements

`edgeEffects`	The most common edge effect corrections

possible values:

`periodic`	wrap-around in all dimensions, i.e. leaving objects enter through the other end (default value)
`reflective`	all borders are reflective, i.e. objects bounce on borders as on rubber walls
`island`	a space with oblivious borders in all directions, i.e. leaving objects are lost forever
`unbounded`	an infinite space adapting to location of items
`bounded`	a space with sticky borders in all directions, i.e. objects that bump into the border stay there
`tubular`	wrap around borders in the first dimension, sticky borders in all other dimensions
`custom`	user-specified border properties – provide a borderType property

`precision`	Spatial resolution (in space units). Two locations are considered identical if their distance is below this value.
`units`	The measurement unit for distances and coordinates in this space
`guardAreaWidth`	The width of a guard area around the space used to define an observation window
`observationWindow`	A rectangle defining which part of the space is visible

Cross-links for `space`:

`coordinate → space:<name>` **{1..*}**

This link tells which descriptors are used as coordinates to locate system components in this space. This link has a required property `rank` which tells to which dimension of space a coordinate refers to. For example, you may set to 1 the rank of the *x* field and to 2 the rank of the *y* field if you want your 2D space coordinates to be *x* and *y*, in this order.

`useRNG → rng:<name>` **{0..1}**

Points to the random number generator channel to use with this space (e.g., to generate random locations). If unset, the default random number channel is used.

3Worlds library of spaces

1. `FlatSurface`: This is a 2 dimensional, continuous space with rectangular limits. Internally, it is coupled to a Quadtree (ref) to optimise the search of nearby points when establishing relations (`relateToDecision` function).

   Additional properties for space type `FlatSurface`:

`x-limits` An `Interval` representing the minimum and maximum values of the x (first) dimension.

`y-limits` An `Interval` representing the minimum and maximum values of the y (second) dimension.

2. `SquareGrid`: This is a 2 dimensional, discrete space with rectangular limits and square cells. The locations are only known at the cell level, which helps optimize searches for nearby locations.

   Additional properties for space type `SquareGrid`:

   `cellSize` The length of the side of the (square) cells in space units

   `x-nCells` The number of cells in the x (first) dimension.

   `y-nCells` The number of cells in the y (second) dimension.

Shapes and topology

This part is still under construction.

The setup of an initial state for a simulation

To run a simulation, an initial population of system component, group, life cycle, and system *instances* must be constructed from their respective *types* (e.g. `groupType`, `componentType`, etc.). They constitute the *initial state* of a simulation. The initial state is kept in memory by the simulator, and is re-loaded at every simulator reset prior to a new simulation run. To set descriptor values in initial instances, two methods are possible: through the use of a specific *function* (`setInitialState`), or through external data files listed as *data sources*.

Component

`/3worlds/…/componentType/component:<name>` {0..*}

Use this node to instantiate components of a given `componentType`.

Properties for `component`:

`nInstances` Number of instances to create. If not set, the default behaviour is to create only one instance of this component. Otherwise, `nInstances` components are created, all with identical descriptors - unless a `setInitialState` function is used that can use random numbers to differentiate instances.

Cross-links for `component`:

`instanceOf → group:<name>` **{0..1}**

This link tells to which `group` these component instances belong. It is required if `groupTypes` have been defined in the model, optional otherwise. This property is ignored if a `loadFrom` link (see below) is present.

Group

`/3worlds/…/groupType/group:<name>` {0..*}

Use this node to instantiate a group of components. Use groups to set properties that are shared by a set of component instances, e.g. a species name or group-level constants.

Cross-links for `group`:

`groupOf → componentType:<name>` **{0..1}**

If no `component` node is defined for this group, this link is required to specify which `componentType` should be used for instantiation of components of this group. In any other case it is not required. You can view this as a replacement for the `instanceOf` link above: either must be present for a group, but not both.

`cycle → lifeCycle:<name>` **{0..1}**

This link tells in which `lifeCycle` instance this group is involved. The `groupType` of this `group` must be compatible with the `lifeCycleType` category requirements.

Life cycle

`/3worlds/…/lifeCycleType/lifeCycle:<name>`

Use this node to instantiate a life cycle applying to a set of groups.

Setting decriptor values

A first method to set descriptor values is to define a `setInitialState` function as a child of the `componentType`, `groupType`, `lifeCycleType` or `system` nodes. Such functions are only called once, at the beginning of a simulation, to setup the *initial state* of a model. In a `setInitialState` function, only drivers and constants can be set (it is the only place where constants can be set).

`/3worlds/…/componentType/initFunction:<name>`

`/3worlds/…/groupType/initFunction:<name>`

`/3worlds/…/lifeCycleType/initFunction:<name>`

`/3worlds/…/system/initFunction:<name>`

Properties for `initFunction`:

`type` The type of function used: only `setInitialState` is possible.

Cross-links for `initFunction`:

`useRNG → rng:<name>` **{0..1}**

Points to the random number generator channel to use in this function. If unset, the default random number channel is used.

A second method to set descriptor values is to import data from an external source. This is done by defining a `dataSource` and linking a `component`, `group`, `lifeCycle` or `system` to it:

Cross-links for `component`, `group`, `lifeCycle`, and `system`

`loadFrom → dataSource:<name>` **{0..1}**

This link tells where to read the initial values of the descriptors from.

3.3.4. The *system/dynamics* node

`/3worlds/system/dynamics:<name>` {1}

This node and its sub-tree contains the description of the processes that will change the state of the system and create its dynamics. It is based on the concepts of *time model*, *ecological process*, and *life cycle*. Internally, the `dynamics` node is the *simulator*, i.e. the object which, when kicked to do so, will make all the computations necessary to run a simulation.

The representation of time

Simulation is about mimicking the dynamics of a real system. Here, dynamics is specified by attaching particular behaviours (called processes) to either categories or relation types. Processes may act at a different rhythm or rate in nature, so we need to have a great flexibility in the way time is represented.

Time line

`/3worlds/system/dynamics/timeLine:<name>` {1}

Every simulator has a reference *time line*. Since different ecological processes may run according to different time models, they must refer to a common time frame for interaction to be possible among them. A `timeLine` defines what kind of time scale and time units can be used in a simulation. In 3Worlds, time is always discrete in the end, so that the selected values of time scale and time unit define the time *grain* of the simulation, i.e. the duration below which events are considered simultaneous. Internally, the `ModelRunner` uses integers to represent time, with 1 = one time grain.

Properties for `timeLine`:

`scale`

This property specifies the type of time scale to use. The usual time units pose many problems, because years, months, weeks and days are not integer multiples of each other. The option is either to use a real calendar time scale – but this is not needed in most simulation studies – or to use approximations which enable year, months, weeks and days to be integer multiples of each other (e.g. an easy approximation is to assume 30-day months, but this means years must be only 360-day long). This property proposes a set of such simplified, compatible sets of units, denoted as time scales.

possible values:

`ARBITRARY`	arbitrary time units with no predefined name (default value)
`GREGORIAN`	real calendar time
`YEAR_365D`	365-days years, no weeks, no months
`YEAR_13M`	28-days months, 13-months/52-weeks years
`WMY`	28-days months, 12-months/48-weeks years
`MONTH_30D`	30-days months, weeks replaced by 15-days fortnights
`YEAR_366D`	366-days year, months replaced by 61-days bi-months
`LONG_TIMES`	long time units only (month or longer), calendar-compatible
`SHORT_TIMES`	short time units only (week or shorter), calendar-compatible
`MONO_UNIT`	single time unit, calendar-compatible

`shortestTimeUnit` The shortest time unit used in this model. Note that the time scale constraints the time units compatible with each other for this property.

possible values:

`UNSPECIFIED`	an arbitrary time unit (default value)
`MICROSECOND`	microsecond
`MILLISECOND`	millisecond = 1000 microseconds
`SECOND`	second = 1000 milliseconds
`MINUTE`	minute = 60 seconds
`HOUR`	hour = 60 minutes
`DAY`	day = 24 hours
`WEEK`	week = 7 days
`FORTNIGHT_15`	French-style fortnight = 15 days
`MONTH_28`	month = 4 weeks of 7 days
`MONTH_30`	month = 30 days
`MONTH`	calendar month (= 1/12 of a calendar year), *i.e.* approx. 30,44 days, but with irregular durations (28,29, 30 or 31 days)
`BIMONTH_61`	2 months = 61 days
`YEAR_336`	year = 12 months of 4 weeks of 7 days
`YEAR_360`	year = 12 months of 30 days
`YEAR_364`	year = 52 weeks of 7 days = 13 months of 28 days
`YEAR_365`	year = 365 days
`YEAR`	calendar year, *i.e.* approx. 365.25 days, but with irregular durations (365 or 366 days)
`YEAR_366`	year = 6 bimonths of 61 days
`DECADE`	decade = 10 years
`CENTURY`	century = 10 decades
`MILLENNIUM`	millennium = 10 centuries

`longestTimeUnit` The longest time unit used in this model. *cf.* `shortestTimeUnit` for valid values

`timeOrigin` The value of time at simulation start, either as a long (number of `shortestTimeUnit`) or as a date; 0 by default.

Timers

`/3worlds/system/dynamics/timeLine/timer:<name>` {1..*}

Ecological processes may be run following different times. A timer is a particular way of representing time in the simulator. Timers may differ in parameters, like e.g. two timers using different time steps; but they can also be radically different in their logic: e.g. clock-like ticking vs. event-driven simulation.

How do timers work?

Internally, each timer is an instance of the class `Timer`, which is able to return the next time where computation must be made according to its own logic. Every iteration of 3Worlds starts with the simulator asking all timers for their next time, then keeping all of those with the next time closest to last time, and executing them *in unpredictable order*. 3Worlds uses long integers to measure time internally, which means that simultaneous events are possible (times may be strictly equal). The finest time interval is given by `TimeLine.shortestTimeUnit`.

Properties for `timer`:

`subclass` the type of `timer` to use.

possible values:

`ClockTimer` Time is incremented by a constant amount *dt*. This is commonly used to simulate regular processes like growth.

`EventTimer` Model dynamics generates *events* and computes the date in the future at which they are going to occur. This is commonly used to generate irregular processes like fire occurrence.

`ScenarioTimer` **Not yet implemented**.

Additional properties when `class = ClockTimer`

`timeUnit` the base time unit used by this timer. *cf.* `timeLine.shortestTimeUnit` for the valid values of this property

`nTimeUnits` the number of base time units in the time unit of this model (*e.g.*, a model may have a 2 year time unit)

`dt` The constant time increment used in this `ClockTimer`, expressed as an integer number of `timer` base unit (= `timer.nTimeUnits` × `timer.timeUnit`). For example, if the `timer` has `timeUnit` = `DAY` and `nTimeUnits` = 3, `dt` is expressed in units of 3 days (e.g. `dt` = 2 means the time increment is 6 days).

if calendar time is used (`timeLine.scale` = `GREGORIAN`), then `dt` will sometimes not be constant (e.g. if `dt` = 2 `MONTH`, `dt` will vary in duration between 59 and 62 days according to the exact date).

`offset` In some cases it is desirable to offset a `ClockTimer` relative to another one running at the same time step, for example to be certain of their order of execution or to simulate cyclic phenomena (e.g. seasons). The timer will start after `offset` × `timeUnit` units, where `offset` is a fraction between 0.0 and 1.0.

Additional cross-links when `class = EventTimer`

An `EventTimer` maintains a queue of time events that gets populated by user-defined ecological functions. This way, future events depend on the dynamics of the system. Functions that can populate an event queue will have an `EventQueue` argument in their heading, usable to create future time events. Events must set a time in the future, i.e. their time must be larger than current time. Event times are always expressed in `Timeline.shortestTimeUnit`s in an `EventTimer`.

`fedBy → initFunction:<name>` **{1}**

This link is used to set the first event of an `EventTimer` at simulation start, hence from an `initFunction`. Without this initial event, the timer will never start. More than one event can be created here.

`fedBy → function:<name>` **{0..*}**

These links indicate which functions can populate the event queue with time events. Multiple events can be created here.

Simulation stopping condition

`/3worlds/system/dynamics/stoppingCondition:<name>` {0..*}

A simulation may be run indefinitely (interactively), but in big simulation experiment it is useful to automatically stop the simulations according to some criterion. Besides the simplest stopping condition, reaching a maximal time value, 3Worlds provides many other possibilities to stop a simulation (e.g. based on a population size, on a variable passing a threshold value, etc.).

When no stopping condition is defined, the simulation will run indefinitely.

Properties for `stoppingCondition`

`subclass` The type of stopping condition to use

possible values:

`SimpleStoppingCondition`
: Simulation stops when a maximal time value is reached.

`ValueStoppingCondition`
: Simulation stops when a variable in a reference system component reaches a given value.

`InRangeStoppingCondition`
: Simulation stops when a variable in a reference system gets within the given range.

`OutRangeStoppingCondition`
: Simulation stops when a variable in a reference system gets out of the given range.

`MultipleOrStoppingCondition`
: Compound stopping condition: simulation stops when *any* of the elementary stopping conditions within this multiple condition's list is true.

`MultipleAndStoppingCondition`
: Compound stopping condition: simulation stops when *all* of the elementary stopping conditions within this multiple condition's list are true.

Additional properties when `class = SimpleStoppingCondition`

`duration` The duration of the simulation beyond time line `timeOrigin`, in time line `shortestTimeUnits`.

Additional cross-links when `class = ValueStoppingCondition`, `InRangeStoppingCondition`, `OutRangeStoppingCondition`

`stopSystem → component:<name>` **{1}**
: The system component in which some variable will be checked to stop the simulation.

Additional properties when `class = ValueStoppingCondition`, `InRangeStoppingCondition`, `OutRangeStoppingCondition`

`stopVariable` The name of the variable in `stopSystem` which values are used to decide to stop the simulation.

Additional properties when `class = ValueStoppingCondition`

`stopValue` The value of `stopVariable` at which to stop the simulation.

Additional properties when `class = InRangeStoppingCondition`, `OutRangeStoppingCondition`

`upper` The upper value of the `stopVariable` range. Only `double` values are accepted.

`lower` The lower value of the `stopVariable` range. Only `double` values are accepted.

Additional cross-links when `class = MultipleOrStoppingCondition`, `MultipleAndStoppingCondition`

`condition → stoppingCondition:<name>` **{1}**

These links point to the stopping conditions that will be used as elementary stopping conditions by the multiple and/or stopping condition. Use these links to construct complex stopping conditions.

The transformations of a system component

Changes in a *system component* through time may be of different kinds: changes in *state*, i.e. in its driver and decorator variables; or more radical changes where the component actually changes *category*, so becomes represented by a different set of variables. Plus, a component may have an ephemeral life (lifespan property), which means component objects are dynamically created or deleted during a simulation.

Process

`/3worlds/system/dynamics/timeLine/timer/process:<name>` {1..*}

Processes are used in 3Worlds to compute change in *system components*. Each process acts on system components of a particular group of categories and is scheduled by a particular timer. Processes contain user-defined code that represents ecological processes. This gives 3Worlds its versatility: one can mix in a single model completely different ecological entities (system components of different categories), implement any ecological process depending on user needs, and put them to work on different time scales (times). A `process` is run according to its parent `timer`.

Cross-links for `process`

A process can act on a single system component at a time (called the *focal* system component), or on a pair of components linked by a relation (called the *focal* and the *other* system components). This is specified using the `appliesTo` cross-link (one at least must be present):

`appliesTo → category:<name>` **{0..*}**

These links indicate the categories of system components that will be acted on by the process.

`appliesTo → relation:<name>` **{0..1}**

This link indicates to which relation type between system component the process applies.

A process **must** apply either to categories (*category process*) or to a single relation type (*relation process*), but cannot apply to both.

`dependsOn → process:<name>` **{0..*}**

This link tells that the process must be activated *after* the processes targeted by the links. Use this link to organize computations when there are dependencies between them.

`inSpace → space:<name>` **{0..1}**

This link indicates that this process will make use of this space for its optimisation of neighbour searches. Use this link when you want fast search for candidates to establish a relation, i.e. in association with the `relateToDecision` function. The required property `searchRadius` in the `inSpace` link tells to which *maximal* distance candidates for establishing a relation must be searched. 3Worlds will only present components at a shorter distance than `searchRadius` (possibly none) to the `relateToDecision` function. A value of zero means the algorithm searches for the closest neighbour only (possibly more than one if all exactly at the same distance), whatever its distance to the focal component.

Forgetting to set the value of `searchRadius` is a common source of unexpected behaviour in spatial models.

How do processes work?

Internally, an instance of the `Process` class implements a '*for each*' loop on system components during an iteration step, as scheduled by its `Timer`. The order in which the components are processed in a loop is unpredictable (not fixed, nor random); the only safe assumption is to consider that they are all processed simultaneously.

If the process applies to a list of categories, it is a single loop on all components belonging to these categories. This is the `ComponentProcess` sub-class.

If it applies to a relation type, it is either (1) a double loop, possibly optimized by using a space, on components belonging to the 'from' list of categories of the relation type, and components belonging to the 'to' list of categories of the relation type (`SearchProcess` sub-class); or (2) a single loop on relations instances of its relation type (`RelationProcess` sub-class).

Processes of a same timer are run in any (= unpredictable) order unless explicitly ordered by the use of the `dependsOn` cross-link.

In short: process = (parallel) loop on components within one time step.

Function

`/3worlds/system/dynamics/timeLine/timer/function:<`*name*`>` {1..*}

This node is used to specify the details of the computations made in a `process`. The `process` defines which system components are going to be activated and at what time in the simulation course; the `function` defines which computations, in detail, will be applied to the system components of that process. This enables to build complex computations applying to one component (a series of `function`s within a `process`) in the context of a particular subset of components (`process`).

There are different types of functions, differing by the way they affect system components and relations. The selection of a function type will trigger the generation in the model java source file of a method having the name specified in the `function` node name. This method is expected to be edited by the modeller in order to implement her/his favourite version of the ecological process modelled by the function.

The name of a function must be a valid java class name, starting with an uppercase letter.

Properties for `function`:

`type` This property specifies which kind of biological function will be implemented within the linked `Process` obect.

possible values:

`ChangeState`	change the state, i.e. the values of the descriptors of a system component (default value)
`ChangeCategoryDecision`	change category of a system component according to life cycle (has no effect if no life cycle is specified)
`CreateOtherDecision`	create another system component, of the same categories if no life cycle is present, otherwise as specified by the life cycle
`DeleteDecision`	delete self
`ChangeOtherState`	*focal* changes the state of *other*
`RelateToDecision`	*focal* establishes a new relation to *other*
`MaintainRelationDecision`	decision to maintain or remove an existing relation
`ChangeRelationState`	change the state of a relation, i.e. possibly both the state of *focal* and *other* at the same time
`SetInitialState`	sets the initial state of a newly created SystemComponent
`SetOtherInitialState`	sets the initial state of a newly created SystemComponent given a parent component

> The `relateToDecision` function has a special status among relation processes as it is used to establish a relation, whereas all other functions for relation processes use an already established relation. As a result, a process parent to a `relateToDecision` function cannot be parent to any other type of function. For computation efficiency, it is recommended to associate a *space* to the process parent to a `relateToDecision` function.

Cross-links for `function`:

`useRNG → rng:<name>` **{0..1}**

Points to the random number generator channel to use in this function. If unset, the default random number channel is used.

Additional properties when `function = createOtherDecision`:

`relateToProduct` A logical value (`false` by default) which tells to establish a permanent `parentTo` relation (defined in the `predefined` sub-tree: Section 3.3.8) between a parent component and its offspring created by calling `createOtherDecision`.

`/3worlds/…/function/snippet:<name>` {0..1}

This node is used in 3Worlds tutorial and test models to store the java code of functions (in its `javaCode` property).

We recommend not to use snippets, or only for simple models, since it may cause problems when the code contained in a snippet contains compile errors, something very easy to achive when, e.g., renaming a field or table. If you use it, do it with caution.

How do functions work?

Functions are called within the loop on components of their parent process. The processing order of functions within their process is constant:

- In a `ComponentProcess`:
  - execute all `changeState` functions
  - then, all `deleteDecision` functions,
  - then, all `createOtherDecision` functions,
  - then, all `changeCategoryDecision` functions,
  - then send data to all data trackers
- In a `SearchProcess`:
  - execute all `relateToDecision` functions
- In a `RelationProcess`
  - execute all `changeOtherState` functions
  - then, all `maintainRelationDecision` functions,
  - then, all `changeRelationState` functions

The order of these computations does not matter for *driver* descriptors, nor for component creation and deletion, because all the changes are postponed until after the process loop is over. For example, a component may in the same time step reproduce an die. The order may only matter for *decorators*, which are provided here as a convenience for complex computations.

A way to make your model code efficient is to group functions run with the same timer in a single process, so that they are called in a single pass on components instead of many in case you use different processes. This assumes that your functions can be processed in any order without consequences on the outcome - which is not always true. If you need a precise ordering of function calls because, e.g. one of them requires that a decorator variable has been set by another, then you must place your functions in different processes linked by a `dependsOn` cross-link to tell which is computed first.

At the end of all process loops (of all timers activated for the current time step):

- all *decorators* are set to zero;
- the values of all *drivers* are updated (i.e. values computed by functions are stored in a *next* driver data structure, which now replaces the *current* driver structure);
- *components* to be deleted and to be created as decided by `-Decision` functions, and *ephemeral* relations to create/delete, are now inserted in the current list of components;
- *automatic* descriptors (e.g. component age, population numbers) are updated;

- *spaces* are updated according to changes in the component community;
- *permanent* relations between components are updated.

Function consequences

`/3worlds/…/process/function/consequence:<name>` {0..*}

Some functions may imply consequences: for example, a decision to delete another system component may be followed by a change in state based on the deleting component's state at the time it is deleted. Such functions that are only activated when certain events take place are called *consequences* and may be specified by a child node to a function. Here also, rules apply:

function	consequence	use to
`changeState`		
`changeRelationState`		
`changeOtherState`		
`deleteDecision`	`changeOtherState`	carry over values to another component linked by a `returnsTo` relation
`createOtherDecision`	`setOtherInitialState`	set the initial state of the new component
`relateToDecision`		
`changeCategoryDecision`	`setOtherInitialState`	carry over and compute values from the former component (*focal*) to the new recruit
`maintainRelationDecision`		
`relateToDecision`		

Consequence functions have the same properties and cross-links as functions (cf. above).

How do consequence functions work?

Internally, consequence functions are just functions which are called *immediately after* their parent function. E.g., in a `ComponentProcess` loop on `deleteDecision` functions, each `deleteDecision` function is immediately followed by a call to its consequence `changeOtherState` function, if any.

Data tracking

`/3worlds/…/process/dataTracker:<name>` {0..*}

Data trackers are used to get output from a simulation. This output can then be redirected either to graphic windows of `ModelRunner` or to a data file for later processing with other software. Using a data tracker is similar to what happens in the real world: you set a sensor into the system you want to monitor and wire it to a data logger which translates the signal into human-readable data. There are different kinds of data trackers according to the output format and the sampling method you want to use; and different ways of wiring these trackers to the simulated system and its components.

A data tracker will record values of descriptors (except constants) of a sample of system components or groups, or of the system itself and send them to any listener (usually a widget) linked to it. The frequency of data sending is determined by the *timer* of the parent *process* of the data tracker.

A `dataTracker` node can only be specified in a *category process*, not in a *relation process*.

Properties for `dataTracker`:

`samplingMode`

This property specifies how to pick system components for data tracking *within a group*. The data will be either (1) selected for one particular system component within each group, or (2) taken from that group's own data, if any (e.g. species population size), or (3) aggregated using some statistical method ; all this, depending on the cross-links of the `dataTracker`. For (1), the default behaviour is that once a system component is selected, it will be tracked until its deletion by the simulator. In all cases, remember that the maximal number of data tracking channels is set by the `sampleSize` property; this property only tells the software how to fit the data coming from possibly many system components into the requested number of output channels.

possible values:

`RANDOM` selects system components randomly to constitute the sample (default value)

`FIRST` selects the first system components of their group (as internally stored) to constitute the sample. Notice that components may come in any order and that this order may change unpredictably during a simulation.

`LAST` selects the last system components of their group (as internally stored) to constitute the sample. Notice that components may come in any order and that this order may change unpredictably during a simulation.

`sampleSize`

How many system components at most will be tracked. Valid values are any positive integer or the keyword `ALL` to sample all the components of a given group.

`statistics` This property lists transformations of the raw data to compute when a group contains more than one system component.

possible values:

`mean`	mean (default value)
`var`	variance
`se`	standard error
`cv`	coefficient of variation (%)
`sum`	sum
`N`	count
`min`	minimum
`max`	maximum

`tableStatistics` This property lists transformations of the raw data to compute in case of a table variable. The grouping is determined by the index specification in the track variable list. cf. property `statistics` for valid values.

`subclass` The type of data tracker to use

possible values:

`DataTracker0D`
: time series tracker; returns (t, x_1, ..., x_n) values (where t is time and x_i the descriptors of interest).

`DataTrackerXY`
: raw tracker; returns (x,y) values at each time step.

`DataTracker2D`
: **unimplemented yet**

Cross-links for `dataTracker`

`trackComponent → component:<name>` **{0..*}**
: The `dataTracker` will track all initial components pointed by these links. If these are *permanent* components, they will be tracked until the end of the simulation. If they are *ephemeral*, **they must all belong to the same *group*** and, when they are deleted, they will be replaced by other components of the same *group* using the sampling strategy defined by the `samplingMode` and `sampleSize` properties.

`trackComponent → group:<name>` **{0..1}**

If the categories of the `process` parent of this `dataTracker` are the same as those of the `groupType` parent of the *group* pointed by this link, then the group data are tracked. If they are the same as the `componentType` of this group's *components*, the `dataTracker` will sample the *group* for components using the sampling strategy defined by the `samplingMode` and `sampleSize` properties.

`trackComponent → groupType:<name>` **{0..1}**

The `dataTracker` will sample the *group type* pointed by this link for *groups* using the sampling strategy defined by the `samplingMode` and `sampleSize` properties.

`trackComponent → system:<name>` **{0..1}**

The `dataTracker` will only track the *system* data.

The items pointed by all the `trackComponent` cross-links must belong to categories compatible with the parent `process` of the `dataTracker` (`process` `appliesTo` cross-links).

The four `trackComponent` cross-links types cannot be used together with the same data tracker.

`trackField → field:<name>` **{0..*}**

The `dataTracker` will track all fields pointed by these cross-links in all objects (*components*, *groups*, or *system*) it is sampling.

`trackTable → table:<name>` **{0..*}**

The `dataTracker` will track all tables pointed by these cross-links in all objects (*components*, *groups*, or *system*) it is sampling.

the trackField or trackTable cross-links must be refer to fields or tables present in the categories of the parent `process` of the `dataTracker` (`process` `appliesTo` cross-links).

Properties for `trackField` *and* `trackTable` *cross-links*:

`index` A list of `String`s giving the range of table indices to track. The following rules are used to specify which cells of the table should be picked:

1. an index specification for a table is enclosed in square brackets: `[]`;
2. vertical bars separate dimensions: e.g. `[||]` refers to a table with three dimensions;
3. a single number is used to specify a single cell in one dimension: e.g. `[1||]` refers to the cell 1 in dimension 1 (Note: this is the *second* cell in this dimension as all dimension indices start at 0 = first cell);
4. a list of cells in one dimension is specified using commas: e.g. `[1|0,2,7|]` refers to the cells 0, 2 and 7 in the second dimension;
5. a range of cells in one dimension is specified using a column: e.g. `[1|0,2,7|2:4]` refers to the cells 2,3 and 4 in the third dimension;
6. a negative number means that all cells in a dimension must be tracked *except* this number: e.g. `[-1|0,2,7|2:4]` indicates that all cells in the first dimension except cell 1 must be tracked.
7. a minus sign in front of a range means that all cells in a dimension must be tracked *except* this whole range: e.g. `[1|0,2,7|-2:4]` indicates that all cells in the third dimension except cells 2, 3 and 4 must be tracked.
8. an empty dimension means all cells in this dimension are to be tracked: e.g. `[1||2:4]` indicates that all cells in the second dimension will be tracked;
9. an empty index string `[]`, or no index string, means that the whole table (all cells) is to be tracked.

The descriptors may be elaborate hierarchical data structure which include a table at some level. For each table in this hierarchy, an index `String` as described above must be given in the index property, in a comma-separated list, e.g.: [0:2|0],[4,6] means that index [0:2|0] should be used for the top-level table of the descriptor, and index [4,6] for the bottom-level table.

[TO DO: a few examples]

3.3.5. The *dataDefinition* node

`/3worlds/dataDefinition` {1}

This node and its sub-tree contains the detailed specifications of descriptors used in a model. This information is used by 3Worlds to generate java code for the data structures that will be used in simulations of this particular model. These data structures are then made available to descendants of 3Worlds functions that users will edit and modify to describe their particular model.

The descriptor tree

Every *system component* of 3Worlds potentially holds four kinds of *descriptors*. Each of these (*drivers*, *decorators*, *constants* and *automatic variables*) is a *tree* of two kinds of data structures: *records* and *tables*. Records contain *fields*, i.e. atomic (i.e. not breakable into smaller parts) pieces of data of different types (numbers, character strings, logical values), each field having a name to access its value (i.e. field = (name,value) pair). Tables are multi-dimensional arrays of values of the same atomic type.

The data tree is constructed by allowing records to contain tables as fields (but not records), and by allowing tables to contain records all made of the same fields (but not tables) instead of atomic types. This gives end-user modellers great flexibility to organise their data into elaborate structures.

Record

`/3worlds/dataDefinition/record:<name>` {0..*}

`/3worlds/…/table/record:<name>` {0..*}

A *record* is a data structure made of *fields* (=(name,value) pairs) or *tables* that represent the *descriptors* of a *category* of *components*. All descriptor trees must start with a record as their root. The record name will be turned into a java class name by the 3Worlds code generator for use in end-user code. A record may be nested in a *table*.

A record cannot be empty, i.e. it **must** have at least one child node (field or table), see below.

Field

`/3worlds/…/record/field:<name>` {0..*}

A *field* is an atomic piece of data accessed by its name within a record.

Properties for `field`:

`type` The data type of this field.

possible values:

`Double`	a double precision floating point number (4.9406564584124654 10^-324 to 1.7976931348623157 10^308, symmetric for negative numbers) with 16 significant digits (default value)
`Integer`	an integer [-2147483648; 2147483647]
`Long`	a long integer [-9223372036854775808; 9223372036854775807]
`Float`	a single precision floating point number (1.40239846 10^-45 to 3.40282347 10^38, symmetric for negative numbers) with 8 significant digits
`Boolean`	a logical value {true, false}
`String`	a text string
`Short`	a short integer [-32768; 32767]
`Char`	a character value (16-bit Unicode = UTF16, i.e. 65535 different values)
`Byte`	a very, very short integer [-128 ; 127]
`Object`	anything else **TODO: should we keep this? it's probably useless**

`description` A long description of the field

`units` Measurement units of field values

`precision` Precision used for display of field values

`interval`	Range of possible values for this field (real number types)
`range`	Range of possible values for this field (integer number types)

Dimensioner

`/3worlds/dataDefinition/dimensioner:<name>` {0..*}

A *dimensioner* is used to define the number of entries in a table.

Properties for `dimensioner`:

`size`	The number of table cells in this dimension.

Table

`/3worlds/…/record/table:<name>` {0..*}

A table is a multi-dimensional data structure containing either atomic data types or *records*, all of the same type.

Properties for `table`:

`dataElementType`	The data type of the elements of the table. cf property `field.type` for valid values.
`description`	A long description of the table
`units`	Measurement units of table values
`precision`	Precision used for display of table values
`interval`	Range of possible values for this table (real number types)
`range`	Range of possible values for this table (integer number types)

These properties are not needed when the table is a table of records (cf. above).

Cross links for `table`:

`sizedBy → dimensioner:<name>` **{1..*}**

This link sets one dimension of a table to be of the length found in the dimensioner `size` property. The required property `rank` in the `sizedBy` edge tells the rank of the dimension. All `sizedBy` links of a table should have a different rank, in increasing order starting from 0 to the number of dimensions -1.

Random number channels

`/3worlds/dataDefinition/rng:<name>` {0..*}

Simulation models make extensive use of random numbers to make decisions on components (e.g. decision to delete a component or to create new ones). 3Worlds defines random number channels, that can be attached to specific functions. These channels can be held constant by using the same seed in a series of simulations, thus enabling to conduct factorial experiments controlling for a source of random numbers or another.

Properties for `rng`:

`algorithm` Algorithm used to produce random numbers

possible values:

- `PCG32` Implementation of a permuted congruential generator with 32-bit output (PGC32 (https://en.wikipedia.org/wiki/Permuted_congruential_generator)); fast (56% faster than `JAVA`) and good quality (default value)
- `JAVA` The default Java random number generator of `java.util.Random`; medium speed, poor quality
- `XSRANDOM` Implementation of a Xorshift random number generator as found here (http://demesos.blogspot.com/2011/09/replacing-java-random-generator.html); very fast (76% faster than `JAVA`), medium quality

`seedSource` How to create the seed of the random number generator

possible values:

- `PSEUDO` The random number seed is produced from a call to a unique instance of `java.util.Random.Random()`. It uses time to the nanosecond to produce a 'very likely to be distinct' seed (default value)
- `NATURAL` The random number seed is taken as an element in a table of 1000 natural random numbers that have been obtained from atmospheric noise. Use the property `tableIndex` to specify which item in this table should be taken for the seed
- `CONSTANT` The random number seed is a constant (either 0 or 1, depending on the algorithm)

`resetTime` When to reset the random number channel seed

possible values:

- `NEVER` The random number seed is never reset after initialisation of the random number channel (default value)
- `ONRUNSTART` The random number seed is reset to its former value for every simulation run (producing the same series of random numbers for every simulation)

`tableIndex` Index into an `array[0..999]` of naturally generated random numbers to act as seeds for resetting

If no random number channel is specified, 3Worlds uses a single default channel with algorithm `XSRANDOM`, seedSource `PSEUDO` and resetTime `NEVER`.

3.3.6. The *experiment* node

`/3worlds/experiment:<name>` {1}

This node and its sub-tree describe the experimental design to run using a given *model* and external data sets. Typically, il will tell `ModelRunner` how many simulations should be run, possibly varying some parameters of the model according to some plan, where data should be read and saved. The name is used to differentiate simulation experiments in a meaningful way.

The default, simplest, simulation experiment is just to run a single simulation of the *baseline* model.

Properties for `experiment`:

`nReplicates` The number of times all elementary treatments must be replicated. Use this only if there is an internal source of variation in simulations, like random numbers. Otherwise all simulations will be strictly identical.

Cross-links for `experiment`:

`baseLine → system:<name>` **{1}**

This link points to a model setup (Section 3.3.2) that will be used as a *base line* simulation. A base line simulation is the equivalent of a control in a real experiment, i.e. a reference case that serves as a basis to which other treatments are compared. Often, the base line is the setup for which data is available to compare simulation outputs to.

Simulation duration

`/3worlds/experiment/timePeriod` {0..*}

The duration of a particular simulation is specified using a `timePeriod` node.

[TO DO: sort out the conflict between this and the stopping conditions]

Properties for `timePeriod`

`start` The starting time of a simulation in `timeLine` shortest time units.

`end` The ending time of a simulation in `timeLine` shortest time units.

Both properties are optional. If none is set, the simulation will start at time 0 and run indefinitely.

Cross-links for `timePeriod`:

`stopOn → stoppingCondition:<name>` **{0..1}**

This link tells how to stop the simulation in case no `end` property is given.

The stopping condition has the priority over the `end` property. **TO DO: check this**

Experimental design

`/3worlds/experiment/design:<name>` {1}

An experimental design specifies the method used to perform the simulations, e.g. number of replicate simulations, treatments as changes in parameter values or initial states, etc. An experimental design can be specified by using standard designs, or by passing a design description file.

For more information on experimental designs for simulation experiments, we recommend reading the documentation of the R software *planor* and *mtk* packages (e.g. these packages could be used to generate design files for use in 3Worlds).

[TO DO: OpenMole integration]

Properties for `design`:

`type` This property specifies an experimental design for the simulation experiment. It only provides basic, standard experimental designs. For more elaborate or specialized designs, use an ad-hoc file description of the design (`file` property, *cf.* below)

> *possible values*:
>
> `singleRun` an experiment consisting of a single simulation run (default value)
>
> `crossFactorial` a cross-factorial experiment based on a limited set of parameter values (factors)

`file` This property gives the name of an experimental design file **TODO : expand on this description**

Experimental treatments

Treatments

`/3worlds/experiment/treatment:<name>` {0..*}

An experimental treatment records a particular set of parameter values and initial state to run a simulation or a series of replicated simulations. It is the basic block of the experiment, just as in real-world experimentation.

Treatements may be specified

- in full detail: this is done by specifying more than one `system` nodes, each `system` being used for a different treatment;
- as (minor) changes relative to the experiment `baseLine`.

NB: not fully implemented yet

Treatments as changes relative to *baseLine*

`/3worlds/experiment/treatment/modelChange` {0..*}

NB: not fully implemented yet

Managing experiment data

Inputs: *dataSource*

`/3worlds/experiment/dataSource:<name>` {0..*}

This node describes a source of data to use to instantiate a model: name of the data source, access method, etc.

Properties for `dataSource`

`file` A valid data file name

`subclass` A data format to use to read that file

> *possible values*:
>
> `CsvFileLoader` comma separated value (https://en.wikipedia.org/wiki/Comma-separated_values) `.csv` text file
>
> `OdfFileLoader` OpenDocument (https://en.wikipedia.org/wiki/OpenDocument) `.ods` spreadsheet

Additional properties when `subclass = CsvFileLoader`

`separator` the field separator used for this `.csv` file (default: tabulation `"\t"`)

Additional properties when `subclass = OdfFileLoader`

`sheet` the name of the spreadsheet to load from this `.ods` file (different spreadsheets in the same `.ods` file must be specified as different `dataSource` nodes). If this property is absent or not set, the first spreadsheet will be loaded.

Additional properties when `subclass = CsvFileLoader`, `OdfFileLoader`

`.csv` and `.ods` file formats both assume the data come in 2 dimensional tables with cross-references between the tables. Table columns must match parameter and driver field or table names. Table rows must match species, stage or system component instances.

The following rules must be respected when preparing the data files:

- The data must not contain any missing value or structural empty cells.
- Empty lines are permitted (they are skipped).
- Text data must not be quoted.
- The first data line of any file or spreadsheet must contain column headers. They must match field names as defined under the `dataDefinition` node (Section 3.3.5).

[TO DO how to handle data hierarchy - DataLabels. cf file configuration-dataIO.adoc for other properties as this is going to change in reimplementation]

Additional sub-tree when `class = CsvFileLoader`, `OdfFileLoader`

`/3worlds/dataIO/dataSource/read:<name>` {0..*}

This node specifies that a particular parameter/driver must be read from the file/spreadsheet. The *name* property of the `read` node must match the parameter/driver name to read. By default, when no `read` node is present, *all* parameters/drivers found in the file/spreadsheet will be read. Use `read` nodes to restrict the number of columns to read in a data source.

`/3worlds/dataIO/dataSource/dim:<name>` {0..*}

This node is used when reading data into tables. The node name must be an integer matching the dimension declared in a table under the `dataDefinition` node. It must then have the following property:

`col` header of the column containing the index values for this dimension.

When `dim` nodes are present, the index values contained in the dimension columns specified in `col` are used to fill a table within the same 3Worlds data structure.

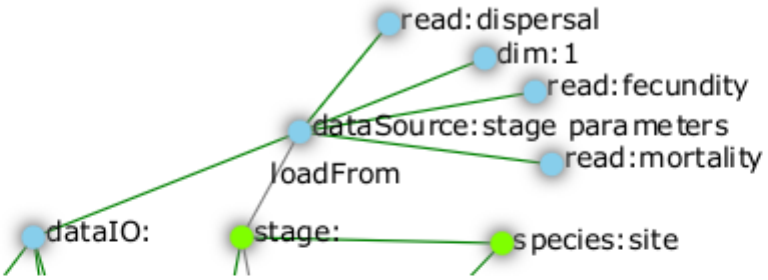
Example of a *dataSource* specification

[TO DO: refactor this example when data sources are reimplemented - this example no longer holds]

This `.csv` file:

stage	dim_1	fecundity	mortality	dispersal	site
population	0	0.1	0.2	0.1	site
population	1	0.1	0.5	0.1	site
population	2	0.2	0.5	0.2	site
population	3	0.5	0.01	0.5	site
population	4	2	0.8	1	site
population	5	3.5	0.56	0.5	site
population	6	14.0	0.02	0.01	site
population	7	2.5	0.001	0.5	site
population	8	6.2	0.03	0.2	site
population	9	3	0.1	0.3	site

with the following specifications:



dataSource:stage parameters#class	fr.ens.biologie.threeWorlds.core.ecology.data.io.OdfFileLoader
dataSource:stage parameters#file	JGSeedDispersalModel.ods
dataSource:stage parameters#idSpecies	site
dataSource:stage parameters#idStage	stage
dataSource:stage parameters#label	dataSource
dataSource:stage parameters#name	stage parameters
dataSource:stage parameters#sheet	stagepar
dim:1#col	dim_1
dim:1#label	dim
dim:1#name	1

Figure 10. Example of a data source configuration.

will result in the instantiation of a single parameter set with species='site', stage='population', and data contained in an 10-cell array of records with 3 fields, fecundity, mortality and dispersal.

How?

- property `idSpecies` states that the column labelled 'site' is the species identifier (last column of the csv file).
- property `idStage` states that the column labelled 'stage' is the stage identifier (first column of the csv file).
- since there is no `idComponent` property, it means this file contains stage parameter data.
- node `dim` with `name` =1 specifies that the data to be read go into a table
- property `col` states that the column labelled `dim_1` contains the indices for dimension 1 of the table.
- the `read` nodes specify that the columns labelled 'fecundity', 'dispersal' and 'mortality' are to be read. Notice that these nodes were not required, since the default behaviour would have caused all these columns to be read anyway.
- finally, the 10 different lines with different table indices (CAUTION: the indices start at 0 for 3Worlds table data structures) will all go into the same parameter set since only one (species name, stage name) pair is given here. Hence only one stage parameter set is instantiated.

Outputs: *dataSink*

`/3worlds/experiment/dataSink:<`*name*`>` {0..*}

This node describes a 'sink' where data resulting from simulation output is to be stored for later processing by other

software.

[Not yet implemented]

Properties for `dataSink`

`file` A valid data file name

Cross-links for `dataSink`:

`source → dataTracker:<name>` **{1}**

This link the origin of the data to save into the sink. Usually, a data tracker

3.3.7. The *userInterface* node

`/3worlds/userInterface:<name>` {1}

This node and its sub-tree specifies the look of the `ModelRunner` user interface. `ModelRunner` is highly configurable and can show many graphs during a simulation run, as help when debugging a new model; or only show a progress bar to improve computing performance when running a big simulation experiment.

3Worlds provides a series of *widgets*, i.e. classes that can manage output from the simulator and either show it graphically or perform some other non-graphic task (e.g. save data to disk). Examples of graphic widgets are: time series or scatter plots, simulation control buttons, progress bars, maps etc... The basic `ModelRunner` GUI just provides places to place widgets: a *top*, a *bottom* and any number of *tabs*. Any number of widgets can be placed within these (Figure 11,Figure 12).

Figure 11. The ModelRunner graphical user interface.

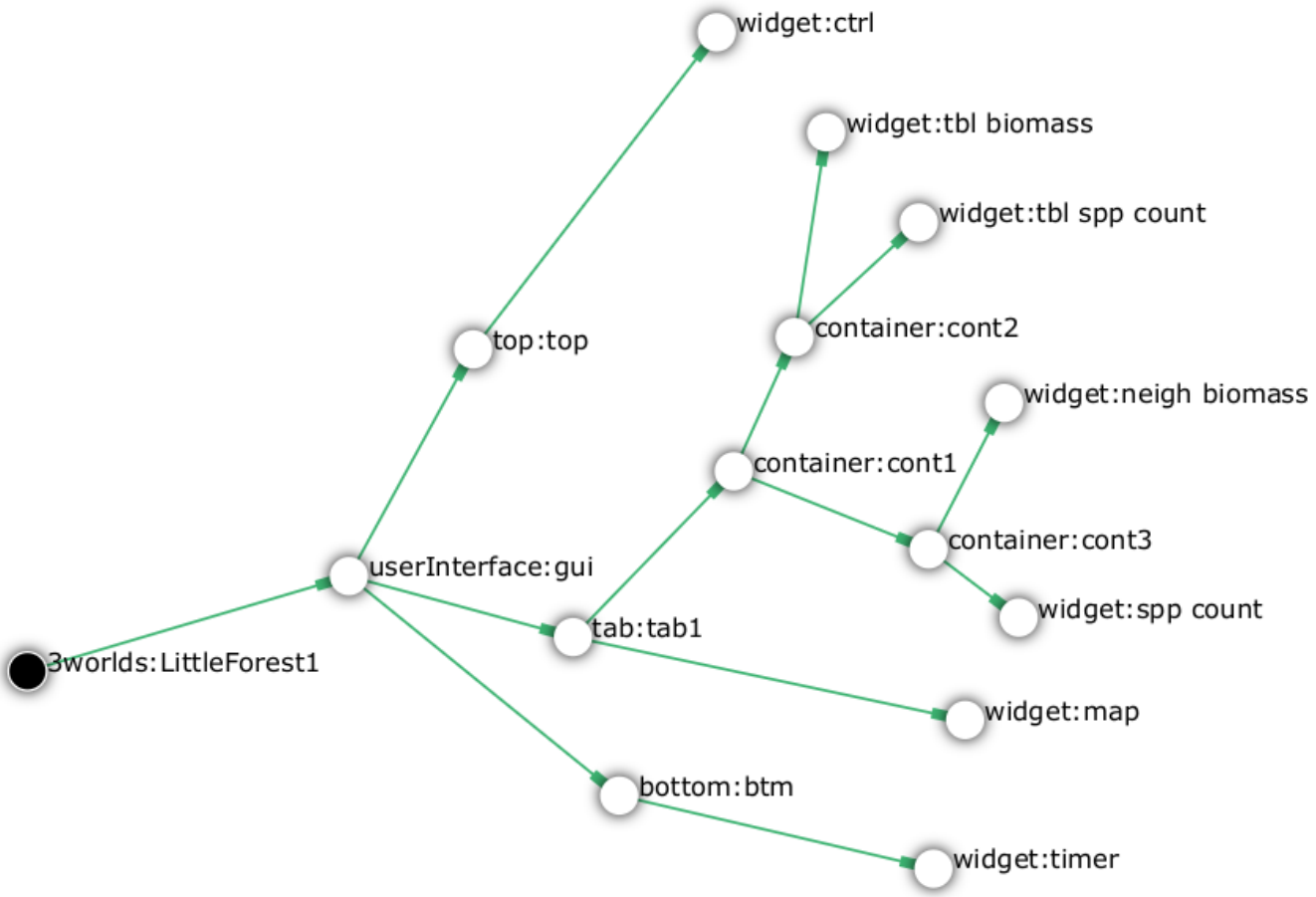


Figure 12. The configuration sub-tree specifying the GUI of Figure 11.

Top

`/3worlds/userInterface/top:<name>` {1}

This is a tool bar and appears at the top, just below the menu bar. Widgets placed here will appear beside each other in an order specified in `ModelMaker` properties (*Order*). The top is typically a place where small widgets are added that remain accessible no matter where other widgets are placed. Typically a controller or progress widgets are placed here. Checks are made by `ModelMaker` to ensure only small widgets can be placed here (Figure 11).

Bottom

`/3worlds/userInterface/bottom:<name>` {0..1}

The *bottom* is a status bar and appears at the bottom of the GUI. It has the same constraints as the *top*. Widgets such as a progress bar can be placed here if desired (Figure 11).

Tabs and containers

`/3worlds/userInterface/tab:<name>` {0..*}

`/3worlds/userInterface/tab/container:<name>` {0..*}

`/3worlds/userInterface/…/container/container:<name>` {0..*}

There can be any number of *Tabs* and each *Tab* can contain any number of widgets through the use of *containers*. *Tabs* are intended for large widgets such as charts and maps. Since only one tab is visible at a time when the model is run, large GUIs can be assembled without the limits imposed by window size (Figure 11). *Tab* contents are structured as a binary tree made of a combination of widgets or containers of widgets. A *Tab* can have:

- One or two widgets;
- One widget and one container; or,
- Two containers.

Containers have the same constraints as tabs - i.e. each container can have:

- One or two widgets;
- One widget and one container; or,
- Two containers.

Only widgets can be leaf nodes in this binary tree i.e you can't have a container or tab that contains nothing.

The widget/container pairs in this binary tree can be arranged vertically or horizontally. This allows all possible arrangements of widgets in the GUI.

Properties for `tab`:

`orientation` This property specifies the way widgets and containers are placed: side by side or on top of each other

> *possible values*:
>
> `horizontal` Display panel is split by a vertical spliter into two panels, first on left and second on right (default value)
>
> `vertical` Display panel is split by a horizontal spliter into two panels, first above second

Properties for `container`:

`orientation` This property specifies the way widgets and containers are placed: side by side or on top of each other

`order` Rank of placement in its container (smaller value: to the left/top; larger: to the right/bottom)

Headless UI

`/3worlds/userInterface/headless` {0..*}

If you want to run a simulation unattended, you may not need a GUI at all. Instead, you may want the simulation to begin immediately and use widgets to write data to disk. Such widgets are call `Headless`. All headless widgets are children of the `gui:Headless:` node.

Widgets

`/3worlds/userInterface/top/widget:<name>` {0..*}

`/3worlds/userInterface/bottom/widget:<name>` {0..*}

`/3worlds/userInterface/tab/widget:<name>` {0..*}

`/3worlds/userInterface/…/container/widget:<name>` {0..*}

`/3worlds/userInterface/headless/widget:<name>` {0..*}

Widgets are the interesting part of the GUI configuration as they provide feedback and control of a simulation. A widget may be added as a child of `top`, `botton`, `tab`, `container` or `headless` nodes.

Properties for `widget`:

`order`	Rank of placement in its container (smaller value: to the left/top; larger: to the right/bottom)

`subclass` A widget class

possible values:

`ControlWidget1`	A simple simulator controller with run/pause, step and reset buttons. Add this as a child of a `top` or `bottom` node. A GUI definition can only have one controller.
`ControlWidget2`	As for `ControlWidget1`, but also displays the stopping condition(s) and the time taken to complete each simulation step and the total simulation time. These timings are only accurate when deploying a single simulation. They are not accurate when deploying multiple simulations. Therefore, this widget is best used when developing a model to check its performance. Use ControlWidget1 otherwise.
`ControlWidget3`	As for `ControlWidget2`, but displays a time series plot of simulation execution time. Add this widget as a child of a `tab` or `container` node.
`ProgressWidget1`	A simple display of simulation time (average time if deploying multiple simulations) and stopping condition. Add this widget as a child of a `top` or `bottom` node.
`ProgressWidget2`	A bar graph of simulation progress intended for use when deploying multiple simulations. Add this widget as a child of a `tab` or `container` node.
`TimeseriesWidget1`	A graph of $x_i = f(t)$, where x_i are component descriptors and t is time. Add as a child of a `tab` or `container` node. This widget uses the fast charting library from Chartfx (https://github.com/GSI-CS-CO/chart-fx). The widget has a number of additional facilites accessed by moving the mouse to the top of the chart display. Here, data can be viewed as a table and exported to file.
`ScatterplotWidget1`	A graph of $y = f(x)$ where x and y are two descriptors of a component. Add as a child of a `tab` or `container` node. This widget also uses the fast charting library from Chartfx (https://github.com/GSI-CS-CO/chart-fx).
`SpaceWidget1`	A map of a space. Add as a child of a `tab` or `container` node.
`TableWidget1`	A table of number values $x_i(t)$. Add as a child of a `tab` or `container` node. Add as a child of a `tab` or `container` node.
`MatrixWidget1`	A colour display of a 2-dimensional table. Add as a child of a `tab` or `container` node. **untested**

`GraphWidget1`	A display of the system graph. Add as a child of a `tab` or `container` node. **not yet implemented**
`HLSimpleTimeSeriesWidget`	Saves time series data to a file. Add as a child of a `headless` node. **untested**
`HLSimpleControlWidget`	As `ControlWidget1`, but for unattended simulations. Add as a child of a `headless` node.

Many widgets have settings which can be adjusted from the *Widget* menu of *ModelRunner*. These settings are automatically saved to a preferences file when *ModelRunner* is closed thus preserving the look and feel of the interface between uses.

Additional properties for widget sub-classes

for `ProgressWidget1`:

`refreshRate`	(optional) The rate at which this widget updates its display (ms) (*default* 250 ms). This property is for the purpose of efficency. Because this widget can display the average progress of a very large number of simulations (e.g 1 million parallel simulations), it would be very inefficient, not to mention uninformative, if the display was updated every time any one of the simulators sent a message.

for `TimeSeriesWidget1` *and* `TableWidget1`:

`nSenders`	(optional) The number of parallel simulators to display (*default* 1). As noted, there can be far too many simulators to reasonably display their output. In such cases, this property limits that number to whatever you think you can handle.
`firstSender`	(optional) The lowest sender (simulator id) to display (*default* 0). Output from simulators is shown for contiguously numbered simulators. This property sets the lowest number from which to display *n* simulator outputs.

for `TimeSeriesWidget1`:

`maxAxes`	Sets the maximum number of *y* axes to display on the chart. As this widget can show time series for any number of variables, it is important to assign them to their own axis. However, if you have so too many outputs the chart will be all axes and no data. There, the number of axes can be constrained with this property.
`bufferSize`	Number of time intervals to plot on the time axis, e.g. a value of 100 will display times 900-1000 when reaching time 1000. That is, what you see is a rolling buffer of data.

for `SpaceWidget1`:

`nViews`	This widget records data produced from any number of simulators. However, its often unreasonable to view every one of what may be a very large number. However, being able to visually compare independent parallel simulations is useful. This setting allow more than one (at the moment this is unlimited) view. A drop-down list for each view allows you to select which data set to display.

Additional cross-links for widget sub-classes

for `ControlWidget2`, `ControlWidget3`, `ProgressWidget1` *and* `ProgressWidget2`:

`trackTime → dynamics:<name>` **{1}**

This link identifies the *dynamics* system the widget is listening to.

for `TimeSeriesWidget1`, `TableWidget1` *and* `HLSimpleTimeSeriesWidget`:

`trackSeries → dataTracker:<name>` **{1}**

This link identifies the `dataTracker` the widget is listening to.

for `SpaceWidget1`:

`trackSpace → space:<name>` **{1}**

This link identifies the `space` the widget is listening to.

for `GraphWidget1`:

`trackSystem → system:<name>` **{1}**

This link identifies the `system` the widget is listening to

3.3.8. The *predefined* sub-tree

`/3worlds/predefined:*categories*` {1}

This sub-tree is not editable. It specifies predefined categories, relation types and automatic variables that are used in any 3Worlds model in assocation with particular objects in the simulator. During the setup of a configuration, ModelMaker will sometimes request you to link some of your nodes to items found in this hierarchy.

Predefined categories are grouped in 3 `categorySet`s:

- `*systemElements*` represent hierarchical levels in the system:
  - `*arena*` is a category attached to the whole system
  - `*component*` is a category attached to any system component
  - `*group*` is a category attached to any group of system components
  - `*life cycle*` is a category attached to any life cycle
  - `*relation*` is currently not in use
- `*organisation*` represent levels of organisation in the system:
  - `nRemoved`, the number of components added to the population during the last simulation step
  - `*assemblage*` is a category attached to populations of components. Members of this category have the following automatic variables:
    - `count`, the population size
    - `nAdded`, the number of components added to the population during the last simulation step
    - `nRemoved`, the number of components removed from the population during the last simulation step
  - `*atomic*` is a category attached to elementary (atomic) components that cannot be further divided
- `*lifespan*` represent the lifespan of a component in the system:
  - `*permanent*`, is a category for components that stay forever

- `*ephemeral*`, is a category for components that can be created or deleted during a simulation.Members of this category have the following automatic variables:
  - `age`, the age of the component
  - `birthDate`, the simulation time when this component was created.

There are two predefined relation types:

- `*parentTo*: *component* → *component*` may be used to relate a component to its offspring, i.e. it enables to track genealogic links between components (see the `CreateOtherDecision` function).
- `*returnsTo*: *component* → *component*` is not yet in use.

3.4. Developing and testing model code

A model of an ecological system in 3Worlds consists in a *configuration file* constructed with the `ModelMaker` application (Section 3) and *java code files* that must be edited by the end-user, here a modeller. Some basic knowledge of java (https://en.wikiversity.org/wiki/Java_Programming/Introduction) is required before going on here.

3.4.1. Generated code: the *model main class*

During the specification of a model, ModelMaker generates java classes meant to be further edited in order to implement the specific behaviours imagined by the modeller. Among these classes, one is meant to be edited; the others are interface code, that you can see as 'glueing' code between the 3Worlds main code and the user-defined one. The file to be edited is named after the 3Worlds root node of the specification file (Section 3.3.1): e.g., if your 3Worlds node has the name `myModel`, the java file to edit will be called `MyModel.java`. In what follows, we call it the *model main class*.

This section gives general rules to follow to successfully edit the generated *model main class* in order for your model to behave as you wish. Section 2.4 describes how to link your `ModelMaker` session with an eclipse development enviromnent where you can edit the generated *model main class*. It is important to read about functions before starting writing your code, in order to master all the options you can setup in `ModelMaker` to specify your function.

For each `function` node of the specification, you will find one static method with the same name in the *model main class*. For example, the **IDHClock** tutorial (Section 5.4) specifies 3 functions, 2 initFunctions and 2 data trackers (Figure 13).

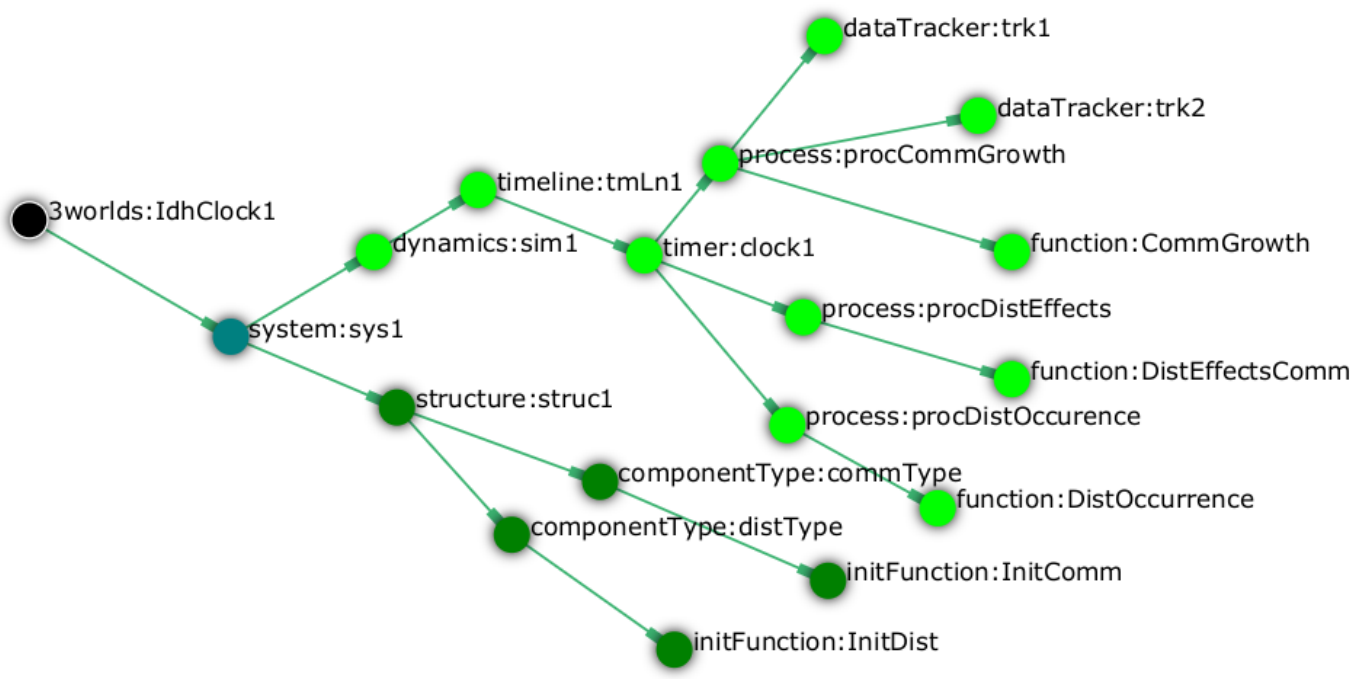


Figure 13. The functions defined in the model of the IDHClock tutorial.

This generates a *model main class* with corresponding methods (code stripped of its javadoc comments):

JAVA

```
package code.sys1;

import static java.lang.Math.*;
import au.edu.anu.rscs.aot.collections.tables.DoubleTable;
import au.edu.anu.twcore.ecosystem.runtime.biology.DecisionFunction;
import java.util.Random;
import code.sys1.generated.*;
// Hey, model developer! You may add your own imports here as needed

public interface IdhClock1 {

	public static void distEffectsComm(
		double t,                              // current time
		double dt,                             // current time step
		int count,                             // whole system autoVar count (#) ϵ[0..*]
		int nAdded,                            // whole system autoVar nAdded (#) ϵ[0..*]
		int nRemoved,                          // whole system autoVar nRemoved (#) ϵ[0..*]
		double freq,                           // focal component constants  ± 0.0 ϵ]-∞,+∞[
		double inten,                          // focal component constants  ± 0.0 ϵ]-∞,+∞[
		DoubleTable other_x,                   // other component drivers population size dim = [40] ± 0.0
		DistEffectsComm.OtherDrv otherDrv,     // next drivers for other component
		double other_div,                      // other component decorators  ± 0.0 ϵ]-∞,+∞[
		DistEffectsComm.OtherDec otherDec,     // new decorators for other component
		DoubleTable other_K,                   // other component constants carrying capacity dim = [40] ±
		DoubleTable other_alpha,               // other component constants interspecific competition coef
		DoubleTable other_r,                   // other component constants growth rate dim = [40] ± 0.0 ϵ
		Random random) {                       // random number generator
	// distEffectsComm ---- Code insert Begin-->
	// distEffectsComm ---- Code insert End----<
	}

	public static boolean distOccurrence(
		double t,                              // current time
		double dt,                             // current time step
		int count,                             // whole system autoVar count (#) ϵ[0..*]
		int nAdded,                            // whole system autoVar nAdded (#) ϵ[0..*]
		int nRemoved,                          // whole system autoVar nRemoved (#) ϵ[0..*]
		double freq,                           // focal component constants  ± 0.0 ϵ]-∞,+∞[
		double inten,                          // focal component constants  ± 0.0 ϵ]-∞,+∞[
		DoubleTable other_x,                   // other component drivers population size dim = [40] ± 0.0
		double other_div,                      // other component decorators  ± 0.0 ϵ]-∞,+∞[
		DoubleTable other_K,                   // other component constants carrying capacity dim = [40] ±
		DoubleTable other_alpha,               // other component constants interspecific competition coef
		DoubleTable other_r,                   // other component constants growth rate dim = [40] ± 0.0 ϵ
		Random random,                         // random number generator
		DecisionFunction decider) {            // decision function
	// distOccurrence ---- Code insert Begin-->
	// distOccurrence ---- Code insert End----<
	}

	public static void initDist(
		double freq,                           // focal component constants  ± 0.0 ϵ]-∞,+∞[
		double inten,                          // focal component constants  ± 0.0 ϵ]-∞,+∞[
		InitDist.FocalCnt focalCnt,            // new constants for focal component
		Random random) {                       // random number generator
	// initDist ---- Code insert Begin-->
	// initDist ---- Code insert End----<
	}

	public static void commGrowth(
		double t,                              // current time
		double dt,                             // current time step
		int count,                             // whole system autoVar count (#) ϵ[0..*]
		int nAdded,                            // whole system autoVar nAdded (#) ϵ[0..*]
		int nRemoved,                          // whole system autoVar nRemoved (#) ϵ[0..*]
		DoubleTable x,                         // focal component drivers population size dim = [40] ± 0.0
		CommGrowth.FocalDrv focalDrv,          // next drivers for focal component
```

```
                double div,                             // focal component decorators  ± 0.0 ϵ]-∞,+∞[
                CommGrowth.FocalDec focalDec,           // new decorators for focal component
                DoubleTable K,                          // focal component constants carrying capacity dim = [40] ±
                DoubleTable alpha,                      // focal component constants interspecific competition coef
                DoubleTable r,                          // focal component constants growth rate dim = [40] ± 0.0 ϵ
                Random random) {                        // random number generator
        // commGrowth ---- Code insert Begin-->
        // commGrowth ---- Code insert End----<
        }

        public static void initComm(
                DoubleTable x,                          // focal component drivers population size dim = [40] ± 0.0
                InitComm.FocalDrv focalDrv,             // next drivers for focal component
                double div,                             // focal component decorators  ± 0.0 ϵ]-∞,+∞[
                DoubleTable K,                          // focal component constants carrying capacity dim = [40] ±
                DoubleTable alpha,                      // focal component constants interspecific competition coef
                DoubleTable r,                          // focal component constants growth rate dim = [40] ± 0.0 ϵ
                InitComm.FocalCnt focalCnt,             // new constants for focal component
                Random random) {                        // random number generator
        // initComm ---- Code insert Begin-->
        // initComm ---- Code insert End----<
        }

}
```

In this example, you can see that:

- the package name `code.sys1` is constructed from the name of the `system` node (Figure 13);
- the interface name `IdhClock1` is constructed from the name of the `3Worlds` node (Figure 13);
- each method name is constructed from a matching `function` or `initFunction` node (Figure 13).
- the argument lists are partly constructed from the `categories` the `process` declaring the `function` applies to;
- the comments documenting the method arguments are constructed from the `field` or `table` node properties (`description`, `precision`, `interval`, `units`, etc.).

Do not neglect these metadata: coding errors due to discrepancies in measurement units between equations are frequent and yield wrong computation results that are difficult to trace. Most publicly available model code does not document the units, and hence is not easily re-usable.

When ModelRunner is launched on the IDHClock tutorial model, it will include the generated `IdhClock1` class and call each of its methods for all system components they are dealing with as specified in the model configuration file.

As you can see in this example, the body of each method is empty, only containing two comments:

JAVA

```
        // initComm ---- Code insert Begin-->
        // initComm ---- Code insert End----<
```

These are the *code insertion markers*. The user-defined code must be inserted between these two lines.

Never remove the *code insertion markers* as they are used by 3Worlds when using code snippets (as in all tutorial and test models).

As *model main class* is a java ***interface***, all data is passed as arguments to its static methods. As you can see in the example above, there may be many arguments. If you look closely, you will see that these arguments match the *descriptors* that were attached to the *categories* to which the *processes* apply. All this information is provided in the

model main class as javadoc comments. For example, the javadoc comment of the `commGrowth` method above produces this output:

commGrowth

```
static void commGrowth(double t, double dt, int count, int nAdded, int nRemoved,
au.edu.anu.rscs.aot.collections.tables.DoubleTable x,
code.sys1.generated.CommGrowth.FocalDrv focalDrv, double div,
code.sys1.generated.CommGrowth.FocalDec focalDec,
au.edu.anu.rscs.aot.collections.tables.DoubleTable K,
au.edu.anu.rscs.aot.collections.tables.DoubleTable alpha,
au.edu.anu.rscs.aot.collections.tables.DoubleTable r, java.util.Random random)
```

CommGrowth method of type *ChangeState*: change the state, i.e. the values of the descriptors of a system component

- applies to categories { *commCat* }

- follows timer *clock1* of type `ClockTimer`, with time unit = 1 t.u

- called before function *distEffectsComm(...)*.

Parameters:

`t` - current time

`dt` - current time step

`count` - whole system autoVar count (#) ϵ[0..*]

`nAdded` - whole system autoVar nAdded (#) ϵ[0..*]

`nRemoved` - whole system autoVar nRemoved (#) ϵ[0..*]

`x` - focal component drivers population size dim = [40] ± 0.0 ϵ]-∞,+∞[

`focalDrv` - next drivers for focal component

`div` - focal component decorators ± 0.0 ϵ]-∞,+∞[

`focalDec` - new decorators for focal component

`K` - focal component constants carrying capacity dim = [40] ± 0.0 ϵ]-∞,+∞[

`alpha` - focal component constants interspecific competition coefficient dim = [40,40] ± 0.0 ϵ]-∞,+∞[

`r` - focal component constants growth rate dim = [40] ± 0.0 ϵ]-∞,+∞[

`random` - random number generator

This comment recalls the categories to which the `commGrowth` method applies, which timer it follows and which time units it uses, and any other useful information like precedence between methods as specified by `dependsOn` cross-links between processes.

Finally, the *model main class* itself has a general javadoc description that gives some information about how to insert useful code into its methods:

Package code.sys1

Interface IdhClock1

`public interface IdhClock1`

Model-specific code for model *IdhClock1*

version demo - Tue Apr 27 15:07:13 CEST 2021

Authors:

Jacques Gignoux
Ian D. Davies

Contacts:

jacques.gignoux@upmc.fr
Ian.Davies@anu.edu.au

Reference publication:

https:/en.wikipedia.org/wiki/Competitive_Lotka%E2%80%93Volterra_equations

Instructions to model developers:

1. Non 3worlds-generated extra methods should be placed in other files linked to the present file through imports.
2. **Do not** alter the code insertion markers. They are used to avoid losing your code when managing this file.
3. For convenience, all the static methods of the `Math` and `Distance` classes are directly accessible here
4. The particular random number stream attached to each `TwFunction` is passed as the *random* argument.
5. For all *Decision*- functions, a *decider* argument is provided to help make decisions out of probabilities. **decider.decide(double)** returns true with probability equal to the argument.
6. For *ChangeCategoryDecision* functions, a *selector* argument is provided to select among different possible outcomes. **selector.select(double...)** returns an integer between 0 and *n* (the number of arguments) using the arguments as weights for probabilities (ie the argument do not need to sum to 1).
7. For *ChangeCategoryDecision* functions, a *recruit* argument is provided that must be used to return the proper category name as a String. **recruit.transition(boolean)** will return the category to recruit to if the argument is true, triggering the change in category of the focal SystemComponent. **recruit.transition(int)** will return the category name matching the index using alphabetical order, 0 index meaning no change in category. For example, if the decision may result in category "young" or "juvenile", 0 will map to no change, 1 to change to juvenile and 2 to change to young.

3.4.2. *Model main class* method arguments

The list of arguments of each method is defined by its function type, the organisation level to which it applies (system, life cycle, group or component), the categories or relation types it applies to, and the user-defined data structures attached to these. Some arguments are read-only, others are writeable so that computation output can be passed back to the 3Worlds main code.

Read-only arguments

Arguments present for all *functions* and *initFunctions*

`random` The random number generator channel associated to this function. For details of how to use an instance of class `java.util.Random`, see the javadoc (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html) for this class. Most of the time, you will be calling `random.nextDouble()` which returns a random double value between 0.0 and 1.0.

Arguments present for all *functions* but not for *initFunctions*

`t` the *current time* passed by the simulator as a double value in units of the `timer` of the parent `process` of the `function`.

`dt` the *current time step*, passed by the simulator as a double value in units of the `timer` of the parent `process` of the `function`.

In the case of multiple timers, for `ClockTimers`, the current time step may be different from the timer's `dt` property because it is the time since last simulator iteration, which may have been triggered by a different timer.

Arguments present for all *'decision' functions*

'Decision' functions are: `ChangeCategoryDecision`, `DeleteDecision`, `CreateOtherDecision`, `RelateToDecision` and `MaintainRelationDecision`. They all return a result that is a decision: a number of components to create (`CreateOtherDecision`), the name of a category (`ChangeCategoryDecision`), or a boolean (all others).

`decider` This argument of class `DecisionFunction` **[TODO: ref to javadoc]** is provided as a helper for transforming probabilities into decisions. This class comprises only one method `decide(…)` which given a probability, returns a `boolean`. More precisely: it returns `true` with the probability passed as argument, ie `decider.decide(0.7)` will return `true` in 7 calls out of 10. It uses the function random number generator (the `random` argument) to make the decision. Technically, this is the realisation of a Bernouilli distribution (https://en.wikipedia.org/wiki/Bernoulli_distribution).

`selector` This argument is only present for the `ChangeCategoryDecision` function type. The `SelectionFunction` class **[TODO: ref to javadoc]**, of which it is an instance, only has one method `select(…)` which, given a list of weights w, returns an integer i with probability $w[i]/\Sigma_i w[j]$, i.e. a realisation of a single trial of a multinomial distribution (https://en.wikipedia.org/wiki/Multinomial_distribution).

`recruit` This argument is only present for the `ChangeCategoryDecision` function type. The `RecruitFunction` class **[TODO: ref to javadoc]**, of which it is an instance, has one method `transition(…)` which returns a category name (`String`), or `null` if the component does not change category. It's argument is either a `boolean` or an `int`, typically the result of a call to `selector.select(…)` or `decider.decide(…)`. Example of use:

```java
public static String recruitSeedling(
    double group_recruitRate,
    ...
    Random random,                          // random number generator
    DecisionFunction decider,               // decision function
    SelectionFunction selector,             // selection function
    RecruitFunction recruit) {              // recruitment function

    return recruit.transition(decider.decide(group_recruitRate));
}
```

Arguments that represent a component in function types which process applies to categories

These function types are `ChangeCategoryDecision`, `ChangeState`, `DeleteDecision`, `CreateOtherDecision`, and `SetInitialState`.

Internally, the system component which is the target of such functions is called *focal*.

The argument list will contain all the fields and tables declared in the root record of the descriptors (drivers, automatic variables, constants and decorators) of the *focal* component. The argument comments will indicate that these arguments are descriptors of the *focal* component, as in this example from the **IDHClock** tutorial for a function of type

`SetInitialState`:

```JAVA
public static void initComm(
    DoubleTable x,                          // focal component drivers population size dim = [40] ± 0.0 ϵ]-∞,+∞[
    double div,                             // focal component decorators  ± 0.0 ϵ]-∞,+∞[
    DoubleTable K,                          // focal component constants carrying capacity dim = [40] ± 0.0 ϵ]-∞,+∞
    DoubleTable alpha,                      // focal component constants interspecific competition coefficient dim
    DoubleTable r,                          // focal component constants growth rate dim = [40] ± 0.0 ϵ]-∞,+∞[
    Random random) {                        // random number generator
    ...
}
```

Arguments that represent the two components of a relation in function types which process applies to relation types

These function types are `ChangeOtherState`, `ChangeRelationState`, `MaintainRelationDecision`, `RelateToDecision`, and `SetOtherInitialState`.

These functions apply to a pair of components linked by a relation. The first of these components (the one at the 'from' end of the relation) is called *focal*, as before. The second one (the one at the 'to' end of the relation) is called *other*.

To distinguish the descriptors of *other* from those of *focal* (since these might belong to the same categories and hence have the same descriptors), all the descriptors of *other* are prefixed with 'other_', as in this example from the **IDHClock** tutorial for a function of `ChangeOtherState` type:

```JAVA
public static void distEffectsComm(
    double t,                               // current time
    double dt,                              // current time step
    double freq,                            // focal component constants  ± 0.0 ϵ]-∞,+∞[
    double inten,                           // focal component constants  ± 0.0 ϵ]-∞,+∞[
    DoubleTable other_x,                    // other component drivers population size dim = [40] ± 0.0 ϵ]-∞,+∞[
    double other_div,                       // other component decorators  ± 0.0 ϵ]-∞,+∞[
    DoubleTable other_K,                    // other component constants carrying capacity dim = [40] ± 0.0 ϵ]-∞,+∞
    DoubleTable other_alpha,                // other component constants interspecific competition coefficient dim
    DoubleTable other_r,                    // other component constants growth rate dim = [40] ± 0.0 ϵ]-∞,+∞[
    Random random) {                        // random number generator    ...
}
```

Arguments that represent the local context of a component

The local context of a component is the part of the system it always sees. The dynamic graph of a simulated system in 3Worlds **always** has:

- an object representing the whole system, called the *arena* ('the place where things happen': Gignoux et al. 2011 (https://doi.org/10.1007%2Fs10021-011-9466-2)). As 3Worlds uses a dynamic graph to represent the whole system (Section 1.2.2), this object actually represents the whole graph. It matches the *system* node in the configuration graph of the `.ugt` file.

And it **may** (0..* multiplicity) have the following other kinds of nodes:

- objects representing the individual entities of the system, which are nodes in the dynamic graph and are just called *components*.
- objects that represent the common properties of a group of components, which are called *groups*.
- objects that represent the transitions that can occur during the life of a component, and are therefore called *life cycles*.

These four types of objects may belong to *categories* and have *descriptors*, and as a consequence can be passed to

functions of a *process* referencing their categories.

With regard to processes and functions, these objects are treated like components, except they cannot establish *relations*; only true components can. This limits the set of functions compatible with them: groups, life cycles and the arena can only be affected by `ChangeState`, `SetInitialState` and `CreateOtherDecision` (not life cycles) function types. When these functions apply to the categories of a `group`, `lifeCycle` or `Arena` object, they are treated as components above, i.e. they become the *focal* 'component' of the user-defined methods.

The arena, groups and life cycles play a particular role in the dynamic graph, as indicated by their names. They also have implicit, 'ontological' relations with system components: a component always knows about its group, life cycle, and arena because they describe part of its own behaviour. As such, they are always accessible as arguments in the function calls of any component.

Just as for the *other* component above, the descriptors of arena, life cycles and groups are prefixed when they appear in a method argument list, with a comment giving more information on the argument, as in this example from the **Palms** show-case model for a function of `CreateOtherDecision` type:

```java
public static double reproduction(
    double t,                           // current time
    double dt,                          // current time step
    int count,                          // whole system autoVar count (#) ϵ[0..*]
    int nAdded,                         // whole system autoVar nAdded (#) ϵ[0..*]
    int nRemoved,                       // whole system autoVar nRemoved (#) ϵ[0..*]
    int lifeCycle_count,                // focal life cycle autoVar count (#) ϵ[0..*]
    int lifeCycle_nAdded,               // focal life cycle autoVar nAdded (#) ϵ[0..*]
    int lifeCycle_nRemoved,             // focal life cycle autoVar nRemoved (#) ϵ[0..*]
    int group_count,                    // focal group autoVar count (#) ϵ[0..*]
    int group_nAdded,                   // focal group autoVar nAdded (#) ϵ[0..*]
    int group_nRemoved,                 // focal group autoVar nRemoved (#) ϵ[0..*]
    double group_aGinc,                 // focal group constants adult BudHeight growth coefficient (m yr-1) ±
    double group_aPdead,                // focal group constants Mortality : int ± 0.01 ϵ[0.0,1.0]
    double group_deadNbLeaves,          // focal group constants Mortality : nbf slope ± 0.01 ϵ[0.0,1.0]
    double group_decay,                 // focal group constants decay ± 0.0 ϵ]-∞,+∞[
    double group_dis,                   // focal group constants dispersal parameter ± 0.001 ϵ[0.0,1.0]
    double group_fec,                   // focal group constants fecundity * # leaves ± 0.1 ϵ[0.0,+∞[
    double group_jPdNNeg,               // focal group constants Adult P(dN=-1) ± 0.01 ϵ[0.0,1.0]
    double group_remanence,             // focal group constants seedling remanenc (y) ± 0.0 ϵ]-∞,+∞[
    double group_slrec0,                // focal group constants rect els to sls alive ± 0.01 ϵ[0.0,1.0]
    double group_slrec1,                // focal group constants rect els to sls dead ± 0.01 ϵ[0.0,1.0]
    double budHt,                       // focal component currentState  ± 0.0 ϵ]-∞,+∞[
    int dead,                           // focal component currentState dead ϵ[MIN_INTEGER..*]
    double nELSeedlings,                // focal component currentState Nb. EL seedlings ± 0.0 ϵ[0.0,+∞[
    int nleaves,                        // focal component currentState nleaves ϵ[0..*]
    double neighbourhoodIndexAdults,    // focal component decorators neighbourhoodIndexAdults ± 0.01 ϵ]-∞,+∞[
    double neighbourhoodIndexJuveniles, // focal component decorators neighbourhoodIndexJuveniles ± 0.0 ϵ]-∞,+∞
    double neighbourhoodIndexMounds,    // focal component decorators neighbourhoodIndexMounds ± 0.0 ϵ]-∞,+∞[
    double neighbourhoodIndexTrees,     // focal component decorators neighbourhoodIndexTrees ± 0.0 ϵ]-∞,+∞[
    boolean sex,                        // focal component constants female?
    double x,                           // focal component constants x spatial coordinate (m) ± 0.1 ϵ[0.0,300.0
    double y,                           // focal component constants y spatial coordinate (m) ± 0.1 ϵ[0.0,300.0
    Random random,                      // random number generator
    DecisionFunction decider) {         // decision function
    ...
}
```

The code generator decides which arguments must be made available to a method based on this hierarchy: **arena** > **life cycle** > **group** > **component**. When the method applies to a component, then all four hierarchical levels are accessible; when it applies to a group, only arena and life cycle descriptors are available, and the group arguments are treated as the *focal* 'component'.

Space data

When a space is attached to a process through a `inSpace` cross-link, the following argument is added to the method list:

`limits` The limits of the space used with this process. This is an immutable object of class `Box` **[TO DO: link to javadoc]**. It returns the coordinates of the lower and upper ends of the space in all its dimensions through the `limits.lowerBound(int i)` and `limits.upperBounds(int i)` methods, where `i` is the dimension index.

Writeable arguments and method return values

User-defined functions are meant to modify the state of the graph, which technically means induce changes in descriptor values and creation/deletion of graph elements, namely components and relations. This cannot be done with read-only arguments.

Method return values

Decision functions all have return values that are interpreted as follows:

function type	return type	return value
`ChangeCategoryDecision`	`String`	`null` or a new `Category`
`DeleteDecision`	`boolean`	`true` if *focal* is to be deleted
`CreateOtherDecision`	`double`	the number of new components to create; the decimal part is interpreted as a probability to create an extra component.
`RelateToDecision`	`boolean`	`true` if *other* is to be related to *focal*
`MaintainRelationDecision`	`boolean`	`true` if the existing relation between *focal* and *other* is maintained

They all have read-only helper arguments. Example from the **IDHClock** tutorial for a `RelateToDecision` function:

JAVA

```
public static boolean distOccurrence(
    double t,                           // current time
    double dt,                          // current time step
    ...
    double freq,                        // focal component constants  ± 0.0 ϵ]-∞,+∞[
    ...
    Random random,                      // random number generator
    DecisionFunction decider) {         // decision function
    // distOccurrence ---- Code insert Begin-->
    return decider.decide(1.0/freq);
    // distOccurrence ---- Code insert End----<
}
```

Method writeable arguments

Change/Set-State functions have the following writeable arguments as output:

function type	writeable arguments
`setInitialState`	*focal* constants & drivers

function type	writeable arguments
`changeState`	*focal* drivers & decorators
	decorators of *arena*, *life cycle* & *group*
`setOtherInitialState`	*other* drivers & constants
`ChangeOtherState`	*other* drivers & decorators
	decorators of *arena*, *life cycle*, *group*, *other life cycle* & *other group*
`ChangeRelationState`	*focal* drivers & decorators
	other drivers & decorators
	decorators of *arena*, *life cycle*, *group*, *other life cycle* & *other group*

These arguments appear in the argument list as specific inner classes instances with self-explained names: `focalDrv`, `focalCnt`, `focalDec`, `groupDec`, `groupCnt`, `arenaDrv`, `otherGroupDrv`, etc... Each of these arguments will contain the same fields as the original data structure it comes from. Any value set in these arguments will be carried back to the component when the method returns. For example, this is the generated code from the **IDHClock** tutorial for a function of type `ChangeOtherState`:

```java
public static void distEffectsComm(
    double t,                           // current time
    double dt,                          // current time step
    int count,                          // whole system autoVar count (#) ϵ[0..*]
    int nAdded,                         // whole system autoVar nAdded (#) ϵ[0..*]
    int nRemoved,                       // whole system autoVar nRemoved (#) ϵ[0..*]
    double freq,                        // focal component constants  ± 0.0 ϵ]-∞,+∞[
    double inten,                       // focal component constants  ± 0.0 ϵ]-∞,+∞[
    DoubleTable other_x,                // other component drivers population size dim = [40] ± 0.0 ϵ]-∞,+∞[
    DistEffectsComm.OtherDrv otherDrv,  // next drivers for other component
    double other_div,                   // other component decorators  ± 0.0 ϵ]-∞,+∞[
    DistEffectsComm.OtherDec otherDec,  // new decorators for other component
    DoubleTable other_K,                // other component constants carrying capacity dim = [40] ± 0.0 ϵ]-∞,+∞
    DoubleTable other_alpha,            // other component constants interspecific competition coefficient dim
    DoubleTable other_r,                // other component constants growth rate dim = [40] ± 0.0 ϵ]-∞,+∞[
    Random random) {                    // random number generator
    ...
}
```

In this example, the writeable arguments are `otherDrv` and `otherDec`. Both of them are of ad-hoc inner classes defined in the generated sub-directory of the eclipse project for **IDHClock** (file `DistEffectsComm.java`):

```java
public class OtherDec {
    public double div;
}

public class OtherDrv {
    public DoubleTable x;
}
```

Here, to set the `div` field of the *other* decorators to 3.2, simply write `otherDec.div=3.2;` in your method code.

As you can see above, the writeable arguments are also present as read-only arguments.

You may also have noticed in the example above that the comment besides `otherDec` is `// new decorators for other component`, while that besides `otherDrv` is `// next drivers for other component`. What does this mean? Well, this is just a reminder that decorators and drivers are not handled in the same way by 3Worlds (cf. categories):

- **Decorator values** are only valid within a time step and are immediately changed, which means that any method writing a decorator value will change it for all other methods called during the same time step. Hence the '**new**' adjective.
- **Driver values** are carried over to the next time step, and thus are subject to a synchronized modification. To keep all component states consistent, the read-only argument for drivers store the values that were set at the previous time step, i.e. they represent the *current* state of all components. When a modification of their value is computed, it is stored in the proper writeable argument that will be copied into a separate driver state representing the *next* state of the component. *This makes sure that all components within a time step are viewed by each other in the same, consistent state.* It is only after all methods on all components of the current time step have been called that 3Worlds will replace the *current* driver state with the *next* driver state. Hence the '**next**' adjective.

Forgetting about the difference of treatment between *decorators* and *drivers* can be the source of **major** but difficult to detect computation errors. Always know what you do!

3.5. Feeding the model with data

4. Simulation reference: running a simulation experiment with ModelRunner

4.1. General concepts

TODO

4.2. Using ModelRunner: software interface and functioning

TODO

4.3. Getting output from a simulation experiment

TODO

5. Sample models and tutorials

5.1. Tutorial 1: Construct and run a simulation for the first time

5.1.1. Introduction

This tutorial explains the procedure for developing and running a simulation with *ModelMaker*. We start by using a simple logistic growth model (https://en.wikipedia.org/wiki/Logistic_map) as a test case and gradually introduce more capabilities over the course of these tutorials as we explore the full range of modelling features available.

In *3Worlds*, the specifications for a simulation model are developed by constructing a graph of nodes and edges and setting their respective properties. The nodes represent elements of interest in the model while the edges show the relationships between them: the entire graph then constituting the model specifications. The simulation can be run as soon as the specifications are complete and Java code has been written to define the behaviour of the functions you have specified.

As you construct the graph, *ModelMaker* is constantly checking and providing feedback on how these specifications comply with an *archetype*. *ModelMaker* guides you through this process by managing the available options at each stage in the graph construction and by providing an updated list of tasks requiring attention.

While this process greatly accelerates model development, a more important aim is to provide a firm basis for model comparison. As all models developed with *ModelMaker* must comply with the *archetype*, structural differences become apparent when comparing specifications. That is, while you can construct any model, the design, being constrained by the *archetype*, will not be arbitrary. Of course, this is not the only difference between models. The code for each function defined by the specifications will be particular but at least these differences are encapsulated at specific places in the model implementation.

The specification graph is both a tree and a directed graph. Aside from the root, each node has a parent and, as we have said, may have additional edges between nodes to define particular relationships. We refer to these edges as *cross-links* to distinguish them from edges that represent *parent-child* relationships.

5.1.2. Installation

Installing *ModelMaker* is very simple: create a directory called `.3w` in your home directory and copy `tw-dep.jar` to it. Then copy `modelMaker.jar` to your home directory. To start *modelMaker*, open a terminal and type: `java -jar modelMaker.jar`. If you receive a message saying "Unable to access jarfile modelMaker.jar", check for misspelling (including upper-case letters) and that the file is executable using the file manager of your operating system.

To compile your own models, the Java Development Kit (JDK) must be installed on your computer.

5.1.3. Creating a new 3Worlds project

1. Start *ModelMaker* as above. If you haven't read any other *3Worlds* documentation, reading "About ModelMaker" from the main menu (`Help → About`) will provide some preliminary context.
2. Create a new project (`Main menu: Projects → New → Templates → Blank`)
3. When prompted enter 'Tut1' (first letter uppercase). A small graph of seven nodes appears in the graph window.

```
FIGURE1 BLANK TEMPLATE
```

The black node (`3worlds:Tut1`) is the *root* of a graph that forms the model specifications. All nodes are identified by a *label:name* pair. The *label* is the type of node, representing its role, while the *name* is a unique identifier. A few node types, of which the root node is one, must have names beginning with an uppercase letter. These nodes are used to generate Java classes when the simulation compiles. The graph is divided into colour-coded sub-trees where different aspects of the specifications are defined.

At this stage you can try out a few basic edit operations:

- Move a node around the graph display by dragging it with the left mouse button;
- Zoom in and out in the graph display using the mouse wheel while holding down the `ctrl` key (or equivalent on your operating system). The focus of the zoom is set by the mouse position;
- Pan the drawing surface by dragging with the left-mouse button anywhere outside a node (assuming the drawing area is larger than the display);
- Change the node and text size with the *element size* slider in the tool bar; and,
- Right-click on a node to see a display a pop-up menu of edit options for that particular node. It is via these popup menus that the graph is constructed.

A list of tasks remaining to be completed appears at the bottom-left of the main window. This list grows and shrinks as the developing graph is checked against the archetype. A check takes place in the background every time the graph structure or its properties change.

At the top-left of the main window are the two property editors - *Properties* and *Selected Properties*. These show properties for the entire graph or for a single selected node (selected by clicking on the node with the left button) respectively. It's here that values are entered for the properties of nodes and edges created in the following steps.

At this stage, there are five properties under the section `Tut1`. These are properties of the root node (`3worlds:Tut1`) intended to provide a limited amount of documentation for this project. If click on the root node and then select the *Selected Properties* editor, an additional non-editable property is shown call 'built-by'. This property is filled out automatically with your computer login name and the creation date of the project. Viewing non-editable properties can be useful at times. These are always shown in the *Selected Properties* editor but for brevity, not in the *Properties* editor.

Also in the tool bar at the top of the graph display are a number of controls that will be explained below. Float the mouse over these controls to see a hint as to their purpose.

Directory structure

All projects are located within the 3Worlds root directory called `.3w`. The dot indicates a hidden directory. To see these files you may need to do what ever is required by your operating system to show hidden files.

The `.3w` directory contains a Jar file called `tw-dep.jar`. This file contains all dependent libraries for *3Worlds*. It's a good idea to keep a backup of this file.

Within `.3w` there is now a sub-directory called something like `project_Tut1_2020-06-07-01-47-44-708`. Every project directory begins with the key word `project`, followed by the name of the project you entered (`Tut1` in this case) and the creation date and time. The purpose of naming project directories in this way is to make it almost impossible to inadvertently overwrite a project.

Inside this directory are a number of text files:

i. `Tut1.utg` is the model configuration graph we're constructing;

ii. `layout.utg` contains information used to display the graph;

iii. `MM.xml` contains the session's preference settings.

iv. `__StateA1` and sequences of similarly named files are temporary files to support the program's undo/redo editing operations. These files are created during a session and deleted when the session ends.

All these files are text files and you can open and inspect them with a simple text editor. However, as they are written by *modelMaker* you should *avoid editing them directly as this will likely cause problems for your project*.

On the other hand, you can safely delete project directories at any time if you wish. If you accidentally delete the project of a currently open session, it will be recreated automatically by *ModelMaker* (apart from the *undo-redo* data). However, if you delete the entire `.3w` directory you will have to restore `tw-dep.jar` from backup.

5.1.4. Creating the specifications

Having created `Tut1`, the task list shows four nodes are required. This list can be dealt with in any order but in general, a logical approach is to build the specifications as follows:

i. **data definition**: the data structures required;

ii. **dynamics**: define how the modelled system evolves over time.

iii. **structure**: define the organisation of components - their roles and relationships.

For the most part, these tutorials will proceed in this order and leave defining the simulation's **user interface** and **experiment** design until last. Each of these sections are organised as colour-coded sub-trees of the graph.

Data definition

The logistic equation we will implement is: *x(t+1) = rx(t)(1-x(t)*. Though simple, it has interesting chaotic behaviour for values of *r* between 3.7 and 4.0. All we need do for the data definition section is to define the parameter *r* and the state variable *x*.

From here on and throughout these tutorials, parameters are called *constants* (data that does not change over the course of a simulation) and state variables are called *drivers* (data that drives the simulation from one time to the next).

1. Right-click on the data definition node (`dataDefinition:dDef`) (pale red) and select `New node → record` from the popup menu. You're then prompted for a name. The default name is `rec1` . Change this to 'cnsts' (constants) and click `ok` . The mouse pointer immediately becomes a cross-hair: *ModelMaker* is asking where to place this node. Move to some place within the graph display and left-click the mouse.

You can name nodes and edges anything you like but accepting the recommended names and edges will make these tutorials easier to follow. Note that *ModelMaker* will prevent naming nodes or edges with duplicate names.

All nodes in the configuration graph are children of some parent (apart from the root node). You can only create nodes by right-clicking on a parent and selecting a child to create from the available options provided by the popup menu. The items in this menu vary according to the possibilities allowed by the *archetype*. This is one way *ModelMaker* ensures the developing configuration conforms with the *archetype*, greatly simplifying an otherwise complex workflow.

2. Create a `field` node as child of `record:cnsts` , name it 'r' and when prompted, set its type as `Double` .

All `fields` (and later `tables`) must be children of some `record` .

3. Create another `record` as child of `dataDefinition:dDef` and name it 'drvs' (drivers).
4. Create a `field` node as child of `record:drvs` , name it 'x' and again set the type to `Double` .

Note that the names 'drvs' and 'cnsts' don't imply any meaning to the specifications - they're just names. Their *roles* as drivers and constants will be defined later.

This is all the data required for this tutorial. The task list has now grown to seven because the roles of this data remains to be defined.

You can tidy up the graph display by clicking the `L` button (re-apply layout) in the tool bar.

```
FIGURE 2 dDefSubTree
```

Dynamics

The `dynamics` sub-tree specifies how the modelled system will evolve over time. It determines the temporal order of function calls, their type, the conditions under which the simulation will stop and what and when data will be tracked for output.

In the present case, the main task is to call the logistic equation a set number of times and present the result from each time step to the equation at the next time step.

The `dynamics` sub-tree (lime green) is a child of the `system` node - the root of the modelled system that defines both

its dynamics and its structure. These nodes are already present in the "Blank" template we started with. The `dynamics` node is the specification of a type of simulator. There can be many simulators (instances) of this specification running in parallel depending on the experimental design.

A dynamics system must have a common definition of the time scale. There are ten different types of time scale available: all of them exact sub-divisions of time except for the Gregorian scale type which implements the standard Gregorian calendar. The default is `ARBITRARY` which is fine for this tutorial.

TO HERE!!!

There is now a new task to add a `timer` node to the `timeline`.

1. Create a `timer` as a child of `timeline`. Here an extra prompt appears asking for the class of the timer: { `ClockTimer`, `EventTimer`, `ScenarioTimer` }. Select `ClockTimer`. This timer class increments time by a constant step during simulation, unless the timeline uses a Gregorian scale in which case irregularities such as leap years are managed.

There is now a new type of task indicating that a property value for the new timer is incorrect: `[Property] ['[Property:dt=0]' does not satisfy '[Property 'dt=0' must be within [1.0; 9.223372036854776E18].']']`. This just means the value of `dt` (delta time) must be >= 1.

5. In the property editor, change `tmr1#dt` to 1, whereupon a new task appears saying the same thing for `tmr1#nTimeUnits` so set that to 1 as well.

`dt` is the time unit size and `nTimeUnits` is the number of time units per simulation step. There are 22 time unit types available ranging from microseconds to millennia. The current default value of `UNSPECIFIED` is fine for this tutorial - here time is just a sequence of steps.

Note that a model can have any number of `timers` using any of the available time steps and time units as long as the time units selected are compatible with the parent `timeline`. The task messages will indicate if this is not the case. Because the specifications allow for more than one system, it follows there can be many dynamics sub-trees, each with their own time system.

A new task has been posted requiring a `process` node.

6. Create a process node as child of `timer:tmr1`.

A `process` defines a set of computations acting on model components driven at the rate of the parent `timer`. A `component` is defined as a unit of simulation. It can be any physical or biology entity represented in the model that has dynamic behaviour (plants, animals, nutrient pools, lakes, the atmosphere or the rhizosphere etc).

Processes can be composed of any number of functions of various types (much more on this later). We need just one function to implement the logistic equation - a `ChangeState` function that takes the current state of a component and calculates the next state.

7. Create a `function` as a child of `process:p1`, name it `Chaos` and select `ChangeState` as its type.

The function type can't be changed after creating node, so if you make a mistake, delete and recreate it (`Delete` from the popup menu or `Undo` from the main menu).

ModelMaker can link to an Integrated Development Environment (IDE) such as *Eclipse*, to write code for these functions. However, in this tutorial the funcion is only one line of code and we can just associate a code snippet with the function without the need to link to an IDE. The snippet will be inserted in the function when the simulation is

compiled.

8. Create a `snippet` node as a child of `function:Chaos`.
9. In the property editor, locate the `snpt1#javaCode` property, click the edit button ('...') and enter the following text: `focalDrv.x = r*x*(1-x);`

Structure

The `structure` sub-tree describes how the modelled system is organised into separate `components` playing particular roles. In an elaborate model, there can be many `components` but in the present case, we need only one, and for convenience, the `system` node can act as this single `component` without the necessity of actually creating a `component` within a `structure` sub-tree.

Here, the component's *role* will be defined as:

- lifetime: *permanent*;
- organisation: *atomic*
- systemElements: *arena*.
- Uses *r* as a constant and *x* as a driver; and,
- `process:p1` applies to it.

The component is *permanent* because it doesn't die; it's *atomic* simply because it is a single indivisable component and it belongs to the *arena*. No matter how many components a model has, exactly one of them must belong to the *arena* category, a unique top level component - it's more or less a global component accessible to all other components.

While this is complicated for such a simple function, later tutorials will show how this arrangement can be a powerful approach to structuring any complex hierarchical dynamic system composed of interacting physical and biological components.

To create this role, we use nodes of the type `categorySet` and `category`. A `categorySet` is a set of mutually exclusive categories. By that we mean a `component` can only be associated with one category of a given categorySet. So for example *permanent* and *ephemeral* are two categories within a set called *lifespan* and obviously, a component can only be one or the other. Categories and CategorySets are recursive: a CategerySet contains Categories and Categories can contain CategorySets without limit.

Apart from the `system` node doubling as a `component`, an additional convenience is provided: a sub-tree of predefined category sets and categories. We use these nodes to define the role described above. To see this sub-tree:

8. Right-click on the root node and select `Collapse → All`.
9. Right-click again on the root node and select `Expand → predefined:*categories*`.
10. Re-apply the layout ('**L**')

The `predefined:*categories*` sub-tree is created with every new project (collapsed by default) and is *immutable* apart from allowing edges to be added between it and other sub-trees.

There are two `record` nodes within this sub-tree for default handling of average population and ephemeral data. Since the single component used here will be neither of these we can ignore this section:

11. Right-click on `predefined:*categories*` and collapse both the `AVPopulation` and `AVEphemeral` sub-trees.
12. Right-click on the root node, expand the `system` and re-apply the layout.

We are now in a position to define the *role* of the `system` node (a.k.a. `component` in this case).

13. Right-click on `system:sys1` and select `New edge → belongsTo → category:*arena*`.
14. Right-click on `system:sys1` and select `New edge → belongsTo → category:*atomic*`.
15. Right-click on `system:sys1` and select `New edge → belongsTo → category:*permanent*`.

The above edits have created three *cross-link* edges. All *cross-links* are red - thin at the *start node* and thick at the *end node*. Unlike parent-child links, they have names. Generally, the names of *cross links* are not much use. They can be hidden by selecting the drop-down list `E text` at the bottom of the Graph display, and selecting `Role`. The relationship can be read as, for example: `system:sys1 belongsTo category:*arena*`.

Finally, we need to apply `process:p` to a category - in this case, the *arena*.

16. Right-click on `process:p1` and select `New edge → appliesTo → category:*arena*`.

We have yet to relate `system:sys1` to '*x*' and '*r*'. We'll leave that for now until it's needed.

There are now two tasks remaining in the task list: the experimental design and the user interface.

Experiment sub-tree

We now specify the simplest possible experiment: a single run of the model.

1. Collpase the `predefined` sub-tree from the root node and re-apply the layout.
2. Create an `experiment` as a child of the root.
3. Create a `design` as a child of `experiment:expt` and when prompted, select the `type` property.

Experimental designs can take many forms including predefined types such as `crossFactorial` or designs read from a file. For now we just use a predefined `type` with its default value of `singleRun`.

User interface sub-tree

The minimum requirement for a user interface is a controller widget: something that can start and stop a simulation. Of course, we'll also need to display the value of *x* with, say, a time series chart. This can be added later after trying out the simulator.

Simulations can also run without any *graphical* user interface - they still must have a user interface but it need not be visible. This situation is called a 'headless' simulator and can be used on unsupervised systems or systems where a user interface is not possible.

When the simulation is run, its graphical user interface has optionally, a toolbar at the top, a status bar at the bottom and any number of tabs containing any number of widgets. The controller must be placed either in the toolbar or status bar. Here we place it in the toolbar.

1. Collapse `experiment:expt` from the root node (we are finished with this sub-tree).
2. Create a `userInterface` as a child of the root.
3. Create a `top` as a child of `userInterface:gui`.
4. Create a `widget` as a child of `top:top1`, name it 'ctrl' and select `SimpleControlWidget1` from the drop-down list as its class.

The model specifications now comply with the archetype and the code has compiled. Save your work (`Ctrl+s`) and the task list will be empty. The **Deploy** button is now enabled and the traffic light has changed from red to green (bottom

left corner of *ModelMaker*).

In addition, the **Document** button has been enabled. Clicking this button generates an ODD template (Overview, Design concepts and Details) (`Tut1.odt`), an established standard for documenting simulation models. This file can be generated anytime the specifications are valid. When you're statisfied with the specifications, make a copy of this file as a basis for the complete documenting of the model.

NB: If you edit the file without making a copy, those edits will be lost whenever the file is regenerated.

Deployment: launching *ModelRunner*

1. Click the `Deploy` button. *ModelMaker* now launches *ModelRunner* to start the run-time application: *ModelRunner*.

At the top of *ModelRunner* are some control buttons to start, step and stop the simulation. This is the `SimpleControlWidget1` we added above in step 4. The *run* button becomes a *pause* button while running and the *stop* button resets the simulator to its starting state.

However, as expected, there's nothing to see so the next step is to add a time series widget. This is an optional requirement so the task list didn't complain about this.

You can move easily between design and execution of the specifications simply by deploying *ModelRunner*, checking the simulation and quitting to return to *ModelMaker*.

To add a time series for *x*:

2. Quit *ModelRunner* and return to *ModelMaker*.
3. Create a `tab` node from the `userInterface:gui` node.
4. Create a `widget` node from `tab:tab1` , name it 'srsx' and select `SimpleTimeSeriesWidget` from the drop-down list.

A new task has been added to the list requiring an edge from this widget to a dataTracker.

For this widget to receive values of *x*, something must post values of *x* to the widget at the same rate as the `Chaos` function is executed. This is the job of a `dataTracker` and it properly belongs in the `dynamics` sub-tree.

5. Create a `dataTracker` as a child of `process:p1` and choose `DataTrackerD0` as it class. This class of data tracker is suitable for scalar data such as *x*.
6. Create an edge from the `dataTracker` to *x* by selecting `New edge → trackField - > field:x` .

Visually, something different happened this time: the edge appeared and then faded away. This is a gesture to indicate that the edge was created but since the end node is not visible (this would be the case if you we following these steps exactly), it fades away to keep the graph display neat and tidy.

1. Create an edge from `widget:srsx` to the new data tracker.
2. Create an edge from `dataTracker:trk1` to a `component` i.e in this case `system.sys1` . A data tracker must not only track some data but also the `component` that uses this data. However, from the task list it can be seen that *x* and *r* are not part of the *role* defined for `system.sys1` . To do this, we categorise *x* as a *driver* and *r* as a *constant*. The appropriate category belonging to `system:sys1` is the `category:*arena*` . Before doing this we can tidy up the display and practice a little fiddling with the collapse/expand functions by showing just the nodes required to define the roles of *x* and *r*:
3. Collapse all sub-trees from the root node.

4. Expand the `predefined:*categories*` sub-tree.
5. Collapse `All` from the `predefined:*categories*` sub-tree.
6. Expand `categorySet:*systemElements*` from `predefined:*categories*` node.
7. Collapse `All` from `categorySet:*systemElements*`.
8. Expand `category:*arena*` from `categorySet:*systemElements*`
9. Finally, expand `dataDefinition:dDef` from the root node and re-apply the layout.

To create the *roles* for *x* and *r*:

1. From `category:*arena*` select `New edge → constants → record:r`
2. Again from `category:*arena*` select `New edge → drivers → record:x`
3. Save the graph (`Ctrl+s`).

The task list should be empty and the simulation can be re-deployed. You can re-generate the ODD again if you wish.

Stopping conditions

If you examine the graph and all its properties, you may notice that there is no indication as to how long the simulation should run. This means that when we run it we should expect it to continue indefinitely. You may or may not want this. If your model contains an unconstrained exponential function, it may eventually crash unless your code takes measures to handle this. You can add a variety of simple or complicated stopping conditions to the `dynamics` node. These will be discussed in later tutorials.

When we first ran this model it had no output. Now that we have a time series chart, displaying data of unlimited length will make the *ModelRunner* fairly unresponsive. If you press the run button and then the stop or pause button of the controller, it may take a while for the model to actually stop running. So for now, it's best to test the simulation with the `Step` button.

6. Deploy *ModelRunner* (saving first if prompted)
7. Click the `step` button a few times. A time series of zeros is shown.
8. Click the `run` button twice in rapid succession. The time (x axis) now reads approximately 30,000 or so depending on the speed of your computer.

The display is still uninteresting because we haven't set an initial value for *x* or parameterised *r*. This can be done in a number of ways but for this tutorial we will add an initialisation function and a code snippet. We will also include a `stoppingCondition`.

9. Quit *ModelRunner* and return to *ModelMaker*.
10. Collapse all sub-trees from the root node and expand just the `system:sys1` sub-tree.
11. Create an `initFunction` as a child of `system:sys1`. As `system:sys1` is the only component in the specifications, it is the node requiring initialisation of *x* and *r*.
12. Create a `snippet` as a child of `initFunction:Init1`.
13. Enter the following two lines in the `snpt2#javaCode` property:
    i. `focalDrv.x = 0.001;`
    ii. `focalCnt.r = 3.7;`

If you make a typo, the task list will show the details of the compile error.

To complete this tutorial, add a simple stopping condition:

14. Create a `stoppingCondition` as a child of `dynamics:sim1`. When prompted, select `SimpleStoppingCondition` from the drop-down list.
15. Select this new node and in the properties editor, set the value of `stCd1#endTime` to 100.
16. Save, re-deploy and run the simulation. You'll now see a time series of the chaos function of 100 time steps.

5.1.5. Graph layouts

An aspect of *ModelMaker* we have only touched on so far, is the graph layout system.

While using a graph to construct model specifications has many advantages, you can quickly become lost in a confusion of nodes and edges. The advantage in using a graph is that the huge number of options possible can be constrained by context. For example, to have a dynamic `process`, it makes sense that it's associated with a particular `timer`, that other processes working at the same rate are associated with the same `timer` and that all timers are coordinated by the one `timeline`. The user interface for problems such as this would be very error-prone if presented say, as a series of dialog boxes.

ModelMaker has a number of features to help arrange the graph display. These fall into three categories: arranging, hiding and resizing.

Arranging: There are four (five planned) layout algorithms currently available in *ModelMaker* of which three make use of the tree structure of the graph to arrange nodes and one is a 'Spring' based algorithms which treat all edges alike, be they *parent-child* or *cross-link* edges. Tree methods are ideal for examining the parent-child structure while the last two are better suited to examining relations between nodes. Tree methods are deterministic while Spring methods are not. Thus, Tree methods are better for maintaining your orientation to the graph but have the disadvantage of not arranging cross-link edges clearly. Spring methods do a better job of this but the resulting arrangement can change with each application of the layout.

Two of the Tree methods produce a radial layout. These are best suited to examining nodes that have many children such as a record with many fields.

All Tree methods allow selecting any node as the root of the tree. This is achieved from the popup menu of each node. When the 're-apply layout' button is pressed ('**L**') while using a Tree method, the root of the tree becomes the root of the graph (`3worlds:Tut1`).

When a project is first created, the default layout is an **orderedTree**. To change to other layouts, use the local popup menu for any node. This layout persists for repeated applications of the layout function until another is chosen.

You can add a random displacement to nodes to help prevent node and edge text overlapping. This setting is applied whenever the layout is re-applied.

Finally, you can of course move a node anywhere within the graph window.

Information hiding: The following operations can help to hide temporally irrelevent information:

- **collapse/expand**: You can hide or show sub-trees from any node from its local popup menu. In addition, all properties of collapsed sub-trees and removed from the property editor;
- **X** Show/hide cross-link edges.
- < Show/hide parent-child edges.
- >| Move all isolated nodes to one side (after re-applying the layout)

- **Show neighbourhood**: With this feature, you can choose to show only nodes within a given path distance from a selected node.
- **A**: Show all nodes. That is, undo the above operation.
- change the node and edge names by hiding or showing either or both roles and names with the `N-text` and `E-text` drop-down lists.

Resizing:

- **Zoom**: Zoom in and out in the graph window with the mouse pointer as the focus point (`Ctrl - mouse wheel`)
- **Pan**: If the drawing surface is larger than the window, you can drag the drawing surface of the graph window (left click outside a node)
- **Node Size**
- **Font size**

As an exercise if you wish, try displaying just the nodes with *cross-links* using the SpringGraph layout. This is a common way to look at just the *cross-link* relationships between nodes. Generally, adding a screen capture of this and a second screen capture of just the relevant *parent-child* relationships make useful additions to the ODD appendix.

1. Expand all sub-trees from the root node.
2. Collapse `record:AVPopulation` and `record:AVEphemeral` from `predefined:*categories*` (these edges are irrelevant here).
3. Hide all parent-child lines ('<').
4. Set isolated nodes to be moved to one side when layout is next applied ('>|').
5. Right-click on any node and select `Apply layout → SpringGraph`.

That's the end of this tutorial. Recreate this project at anytime from the main menu (`Project → New → Tutorials → 1 Logistic`).

5.1.6. Next

The next tutorial (Tutorial 2) will demonstrate linking this project to an IDE and adding some Java program code.

5.2. Tutorial 2: Linking a *3Worlds* project to a Integrated Development Environment (IDE)

5.2.1. Introduction

The previous tutorial used code snippets to insert Java code into the model specifications. In this tutorial we show how to use an Integrated Development Environment (IDE) to achieve the same end. Generally, the only way to develop any but the simplest of models is to code in a professional IDE such as *Eclipse*, *Intellij* or *NetBeans*. At the time of writing, linking specifications to a Java project has only be tested with the *Eclipse* IDE.

5.2.2. New Java project

First create a Java project in *Eclipse*.

1. In *Eclipse*, create an new Java project and name it `tut2` (first letter lowercase).
2. Right-click on the new project and select `properties → Java Build Path`.
3. Under the `Libraries` tab, highlight `Classpath` and click the `Add External JARs…` button to the right.
4. Navigate to the .3w directory and select `tw-dep.jar`.

5. Click `Apply and close`.

This installs the *3Worlds* dependencies for this java project.

5.2.3. Specifications

Now create a *3Worlds* project using tutorial 1 as a starting point.

1. From *ModelMaker* main menu select `New → Tutorials → 1 Logistic` name it `Tut2`
2. Right-click on the root node (`3worlds:Tut2`) and select `Collapse → All`.
3. Right-click again on the root node and select `Expand → system:sys1`
4. Re-apply the layout (**L**).
5. Delete `snippet:snpt1` and `snippet:snpt2` as they're no longer required.
6. Save (`Ctrl+s`)

Next link this project to the Java project created above.

1. From the *ModelMaker* main menu, select `Edit → Java project → Connect…`.
2. Navigate to your *Eclipse* workspace and select `tut2`.
3. In the *Eclipse* Project Explorer view, right-click on `tut2` and select `Refresh`.
4. In the *Eclipse* main menu, select `Project - > clean` and apply to `tut2`.

While editing in *Eclipse*, a task message may appear in *ModelMaker* from time to time complaining about file dates. If so, make sure your Java files are saved, refreshed and the project clean. The message will disappear when the project is re-complied in *ModelMaker* using the button in the bottom left-hand corner.

5.2.4. Writing Java code

In the `src` folder of `tut2`, you'll now find a new package called `code` and `code.sys1`. These have been generated by *ModelMaker* upon linking. In the `code` package is a Java file with the same name as the 3Worlds project: `Tut2.java`. This single class is where you add all your code. The other classes in `code.sys1` are not intended to be edited. If you do edit these, your edits will be overwritten whenever the specifications are edited in *ModelMaker*.

1. Open `Tut2.java` in the *Eclipse* editor.

Near the bottom of the file are two methods: `chaos` and `init1`, mapped from the names of the function and initFunction nodes we created in `Tut2` with *ModelMaker*. Within the body of these methods are a pair of insertion markers:

- `Code insert Begin-- >`; and
- `Code insert End---- <`.
    1. Add the following code between these markers, taking care to leave the text of the markers themselves unchanged.

`chaos`:

```java
        focalDrv.x = r*x*(1-x);
```

`init1`:

```
        focalDrv.x = 0.001;
        focalCnt.r = 3.7;
```

Because these two methods are mapped to this generated file from the specifications created in *ModelMaker*, it follows that changes to the specifications may result in changes to these methods. So the question arises, how does *ModelMaker* manage this without losing the Java code you write?

If there have been changes to the number, name or type of `function` used in the specification, changes to anything in the `dataDefinition` sub-tree, or changes to the *roles* of `components`, *ModelMaker* will back up your old file as a text file with a unique name (e.g `_Tut2_1.txt`, `_Tut2_2.txt` etc...) within the current java package. This way you can copy and paste you code back to the proper place. Otherwise _ModelMaker_ will never overwrite this file.

When *ModelMaker* searches for differences between previous and current versions of this Java file, it will ignore any comments and import statements and the code you have added between the insertion markers. Therefore, although you can do as you please with the generated comments, you should take care not to alter the text of the markers themselves.

If the task list is empty, you can now execute the specifications by deploying them from *ModelMaker*.

Running the simulation directly from *ModelMaker* is a convenient way to speed up the turn-around time in developing and testing specifications. Once the specifications are stable, you can run your model from a jar: the jar file that is created in your project directory in this case `Tut2.jar`. To do this:

2. Open a terminal, change to your project directory and enter the following text: `java -jar Tut1.jar`

Finally we discuss how to debug your equations by running your model from the IDE.

Debugging your code

Once the specifications are valid (i.e no tasks in the task list) you can quit *ModelMaker* and debug your code from the IDE.

In the default package of the linked java project (`tut2`) is a java file called `UserCodeRunner.java`. This was created the first time you linked the project made with *ModelMaker* to the Java project. By placing breakpoints in the methods you added to `Tut2.java` above, you can debug by running from this `main` class. The correct args have already been provided.

5.2.5. Next

The next tutorial will introduce the use of tables and some additional output `widgets` to add to the simulator's user interface.

5.3. Tutorial 3: Using tables

5.3.1. Introduction

This tutorial introduces the use of tables in *ModelMaker* by implementing a multi-species competitive Lotka-Volterra equation (CLV).

The equation has three *constants* and one *driver*; all of which are tables:

- *K*: carrying capacity
- *r*: growth rate
- *alpha*: interspecific competition coefficient; and a driver,

- *x*: population size.

5.3.2. Specifications

Apart from the use of tables and a more elaborate model equation, these specifications are very similar to those of Tutorial 1. *ModelMaker* provides a library of model templates for common model patterns such this. One of these, the `SimpleClock` template, has already defined a system which is *permanent*, *atomic*, belongs to the *arena*, has a 'clock' timer driving a single process and function, a simple experimental design and a basic user interface.

1. Start *ModelMaker* and create a new: `Projects → New → Templates → 2 SimpleClock` and name the project 'Tut3'.

Data definition

Data is defined through the recursive use of tables and records: records can contain tables and tables can contain records without limit. Records can also contain fields of any primitive Java type. Tables can also contain Java primitive types as well as user-defined records. Tables can have any number of dimensions.

There is usually more than one way of defining the data for any model. For the CLV, we will use a 2-dimensional table for *alpha*, a 1-dimensional table for *r* and *K* and another 1-dimensional table for the population *x*.

2. Collapse all sub-trees from the root node and then expand `dataDefinition:dDef`.
3. Delete `record:decs`. This record was provided by the template and is not needed here.
4. Create a `table` as a child of `record:cnsts`, name it 'alpha', choose `dataElementType` and then set its type to `Double` (the default).
5. Create a second `table` as child of `record:cnsts`, name it 'r' and choose `dataElementType` and then set its type to `Double`.
6. Create a second `table` as child of `record:cnsts`, name it 'K' and choose `dataElementType` and then set its type to `Double`.
7. Create a `table` as child of `record:drvs`, name it 'x', choose `dataElementType` and set its type to `Double`.

To complete the definition, we must set the table dimensions and indexing order for the 2-dim table 'alpha'.

Only one dimensioner is needed: the number of species. This is done by creating a single `dimensioner` node to be associated with all these tables.

8. Create a `dimensioner` as a child of `dataDefinition:dtDef` and name it 'nspp' (number of species).

A new task appears in the task list saying the 'size' property must be greater than zero.

9. Click on `dimensioner:nspp` and go to the `Selected properties` tab. Set a value of `npp#size` to 4 (four species).

Finally, locate the `description` property for each of these tables and add the text from the introduction above. This text, together with other property meta-data such as `range` and `units`, appears as part of the Java documentation when generating Java code from *ModelMaker*. These descriptions are also used when generating the ODD document.

There are now four tasks in the task list indicating we have yet to associate the dimensioner with the four tables. *K*, *r* and *x* have one dimension and *alpha* has two.

10. Right-click on each table node in turn and select `New edge → dimensioner:nspp`. Repeat this for `table:alpha` to get the second dimension.

Now set the rank order for the indexing of `table:alpha`.

11. Click on `table:alpha` and examine its properties in the `Selected properties` editor.
12. Set the value of one of its rank properties to 1 - it doesn't matter which.

Dynamics

Although the task list is now empty (i.e. the specifications are now deployable), we have yet to write the model equation and provide some output. We can now rename the function provided by the template to something meaningful in the present context and add a data tracker to follow *x*.

1. Expand the `system.sys1` sub-tree from the root node and re-apply the layout.
2. Right-click on `Function:F`, select `Rename node` and name it 'Growth'. If the prompt is unresponsive, remember to begin the name with an upper case letter.
3. Create a `dataTracker` as a child of `process:p` and select `DataTrackerD0` as the type (the default).
4. Create an edge from the data tracker to *x*: `trackTable → table:x`.

There are now two tasks in the list, one about 'indexing' and the other asking which 'component' to track. As with the previous tutorials, the component to track is the `system.sys1` node.

4. Create the edge `trackComponent → system:sys1` from the data tracker node.

As we are using a scalar data tracker (`DataTrackerD0`) to follow a table, we must specify which elements of `table:x` to track. Indexing is a property of the edge between data tracker and the table. Edge properties appear in the property list of nodes that are at the start of the edge (thin line) - in this case `dataTracker:trk1`.

5. Click on `dataTracker:trk1` and in the *Selected properites* tab, edit the `trks#index` property by clicking the editor button (**...**). The prompt shows the range of this table - here [0:3] being 4 elements. We can select any or all of these elements. For now choose all of them ('[0:3]'). Check the validity of the entered expression with the 'validate' button.

This indexing will provide four data outputs. Indexing statements can select any number of contiguous or discontiguous table elements. The syntax is the same as that found in the 'R` statisical software.

User interface

We can now add some additional widgets to the user interface as the template provided only a controller.

1. Hide all nodes and expand the `userInterface:gui` node.
2. Add a `tab` as a child of `userInterface:gui`.
3. Add a `widget` as a child of `tab:tab1`, name it 'srsx' and select `SimpleTimeSeriesWidget` as the widget class.
4. Add a second `widget` as a child of `tab:tab1`, name it 'tblx' and select `SimpleDM0Widget`.

Both these widgets are compatible with this data tracker class: the `SimpleTimeSeriesWidget` produces a chart while the `SimpleDM0Widget` displays the data as a continuously updated table. The task list requires these widgets to be connected to a data tracker.

5. Right-click on each of these widgets in turn and select `New edge → trackSeries → dataTracker:trks`.

Save the specifications (`Ctrl+s`) and they're now ready to run. The next step is to create a Java project to write the 'Growth' and 'Init1' functions, the latter having been provided by the `SimpleClock` template we started this tutorial with.

Link to a Java project

1. Follow the steps in Tutorial 2 to create a Java project with *Eclipse* and name it 'tut3'.
2. Link it to this *ModelMaker* project.
3. Open `Tut3.java` in the *Eclipse* editor and enter the following source code between the relevant insertion markers:

`init`:

```
JAVA
for (int i = 0; i < r.size(0); i++) {
    focalCnt.r.setByInt(random.nextDouble() * 2.0, i);
        focalCnt.K.setByInt(0.2 + random.nextDouble(), i);
        for (int j = 0; j < alpha.size(1); j++) {
            if (i == j)
                    focalCnt.alpha.setByInt(1.0, i, j);
                else
                    focalCnt.alpha.setByInt(random.nextDouble(), i, j);
        }
}
for (int i = 0;i<x.size(0); i++)
    focalDrv.x.setByInt(0.2, i);
```

The above method simply initialises the equation constants: growth rate (*r*), carrying capacity (*K*) and the interspecific competition coefficient (*alpha*)to random values and the population size (*x*) to 0.2.

There is a default random number generator (RNG) available to all functions. In later tutorials we will show how the specifications can factor any number of RNGs into groups. For example, one RNG can be assigned to functions of a particuar type such as those effecting reproduction or mortality.

ModelMaker has two types of RNG classes in addition to the standard Java RNG. These two are faster and produce streams of higher quality than the standard Java RNG. There are also various ways of seeding RNGs to ensure their uniqueness and to help with debugging.

'growth`:

```
JAVA
double integrationStep = 0.01;
double[] dxdt = new double[x.size(0)];
for (int i = 0; i < x.size(0); i++) {
    double sum = 0;
        for (int j = 0; j < alpha.size(1); j++)
            sum += alpha.getByInt(i, j) * x.getByInt(j);
        dxdt[i] = r.getByInt(i) * x.getByInt(i) * (1 - sum / K.getByInt(i));
        }
for (int i = 0; i < dxdt.length; i++)
    focalDrv.x.setByInt(x.getByInt(i) + dxdt[i] * dt * integrationStep, i);
```

The model is now ready to run. However, you may want to change the time duration of the simulation from the template default of 100 to 1,000 steps.

5.3.3. Next

The next two tutorials (4 & 5) begin to elaborate the Lotka-Volterra model by developing the `structure` sub-tree through the addition of a disturbance component.

5.4. Tutorial 4: Elaborating the model structure: Testing the Intermediate Disturbance Hypothesis

5.4.1. Introduction

In this tutorial we add a disturbance to effect a community of species modelled with the competitive Lotka-Volterra

equation (CLV) from Tutorial 3. This model could be used to examine the Intermediate Disturbance Hypothesis (IDH).

5.4.2. Specifications

Begin this tutorial using the Lotka-Volterra specifications from the `Tutorials` menu as a starting point.

1. Create a new project from on `Tutorials → 2 LotkaVolterra` and name it 'Tut4'.

Data definition

1. Collapse all nodes except for the `dataDefinition` sub-tree, hide the *cross-links* ('**X**' button), and re-apply the layout ('**L**' button).
2. Add a new `record` as a child of `dataDefinition:dDef` and name it 'decs'.

Data is divided into three classes in the *3Worlds archetype*:

i. *constants* (cnsts),

ii. *drivers* (drvs) and

iii. *decorators* (decs).

Decorators are values derived from the current value of the drivers. They are set to zero at the beginning of each time step. Here, the *decorator* will be a species diversity index (Shannon's), derived from relative abundance of each species.

As noted previously, these names are just for clarity - they don't impose a *role*. You can name a node any thing you like as long as the names are unique. Here, the records are named to indicate their intended roles to keep things organised. The *role* is formally defined when these records are later collected together within a `category` that `belongsTo` a `component`.

3. Add a `field` as a child of `record:decs`, name it 'div' (diversity index) of type `Double`.

We now define some data for the simple disturbance model: frequency and intensity.

4. Add a new `record` to `dataDefinition:dDef` and name it 'distCnsts' (disturbance constants).
5. Add two fields to this record, both of type `Double` called 'freq' and 'inten' respectively.

The records carried over from the previous tutorial should now be given better names. A trick with the Tree layout functions is that they order the trees alphabetically. To arrange nodes so that related nodes of a particular role are close together, prefix the name with the same string - in this case 'dist'. We should also prefix all the data associated with the CLV with 'comm' (community).

6. Rename `record:cnsts` to `record:commCnsts`, `record:drvs` to `record:commDrvs` and `record:decs` to `record:commDecs`.
7. Re-apply the layout. The data fields should now be in a more sensible order.

Fill out the meta-data properties with at least the description of the data as these descriptions will be incorporated in the comments of the generated Java code.

8. Add the following descriptions to the new data definitions properties:
   - `freq#description`: 'average return time'
   - `inten#description:'% population decrease'
   - `div#description:` 'Shannon diversity'

For simplicity, we define intensity as a constant rather than a driver.

That's all the data needed for this tutorial and the task list should now be empty.

Dynamics

In this section we add two new functions: one to initiate a disturbance and second to apply the disturbance to the community.

1. Collapse all sub-trees and expand `system:sys1`.
2. Create a `process` as child of `timer:clock1` and name it 'procDistOccurrance`.
3. Create a `function` as child of `process:procDistOccurrence` and name it 'DistOccurrence' and set its type as `RelateToDecision`.
4. Create a second `process` as child of `timer:clock1` and name it 'procDistEffects'.
5. Create a `function` as child of `process:procDistEffects`, name it 'DistEffects' and set its type to `ChangeOtherState`.

The name the growth process and function should now be changed to something more appropriate:

6. Rename `process:p1` to `process:procCommGrowth`.
7. Rename `function:Growth` to `function:CommGrowth`.

`function:CommGrowth` will perform the same task as it did in the previous tutorial: calculate the next state of *x* from its current state.

`function:DistOccurrence`, a `RelateToDecision` function, will decide if a disturbance is to occur or not.

If a disturbance occurs, `function:DistEffects` will implement the consequences of the disturbance on the community.

This implies an order of execution of these three functions. As all these functions are driven by the same `timer` and therefore occur simultaneously, it means we must make the calling order explicit. To do this we add a `dependsOn` edge between the relevant processes.

First, set `process:procDistEffects` to depend on `process:procDistOccurrence`.

8. Right-click on `process:procDistEffects` and select `New edge → dependsOn → process:procDistOccurrence`.

To force the disturbance effects to take place before the community growth, make `process:procCommGrowth` depend on `process:procDistEffects`.

9. Right-click on `process:procCommGrowth` and select `New edge → dependsOn → process:procDistEffects`.

That's all that's required in this section. The task list indicates that the new processes must belong to some categories. To do that we need to move on to the structure of the specifications.

Structure

Recall from the previous tutorial, that `system:sys1` is acting as a single component. It's *role* was defined as:

i. *permanent*;

ii. *atomic*;

iii. belonging to the *arena*;

iv. accessing particular data from the `dataDefinition` sub-tree; and,

v. has `process:p1` applied to it.

The *arena* is a special category. It must exist in every specification. Any data associated with it (*constants*, *drivers* and *decorators*) is available to all `functions`: the data is global.

The *role* of a `component` is defined by its `componentType`. We'll need two `ComponentTypes`; one for the community (the CLV equation) and one for the disturbance. We also need a `relationType` to define the association between the disturbance and the effected community.

1. Create a `structure` as child of `system:sys1`.
2. Create a `componentType` as a child of `structure:struc1` and name it 'distType'.
3. Create a `component` as child of `componentType:distType` and name it 'dist'.
4. Create a second `componentType` as child of `structure:struc1` and name it 'commType'.
5. Create a `component` as child of `componentType:commType` and name it 'comm'.
6. Create a `relationType` as child of `structure:struc1` and name it 'distEffectComm'.

Before proceeding to the definition of the *roles* of these component types, we should redefine the *role* of `system:sys1` that we inherited from Tutorial 3 when starting this project.

`system:sys1`, in aliasing as a `componentType`, must now be redefined as an `assemblage` with no functions or data associated with it.

First, arrange the graph display so only relevant parts are displayed. These instructions are as tedious to write as they are to follow, but its worth it.

7. Collapse all nodes to the root node.
8. Expand all nodes from the root node.
9. Collapse `experiment:expt` and `userInterface:gui` sub-trees to the root.
10. Collapse `dimensioner:nspp` to `dataDefinition:dDef`.
11. If you want, you can collapse all `fields` and `tables` into their respective tables.
12. Collapse `record:AVPopulation` and `record:AVEphemeral` to `predefined:*categories*`.
13. Collapse `category:*group*`, `category:*space*`, `category:*relation*` and `category:*lifecycle*` to `categorySet:*systemElements*`.
14. Collapse `dynamics:sim1` from `system.sys1`.
15. Re-apply the layout.

Redefine the *role* of `system:sys1` as an `assemblage`.

16. Right-click on `system:sys1` and select `Delete edge → belongsTo → category:*atomic*`.
17. Right-click again on `system:sys1` and select `New edge → belongsTo → category:*assemblage*`.

Remove the data associated with the *arena*.

18. Right-click on `category:*arena*` and select `Delete edge → drivers → record:commDrvs`.

19. Right-click again on `category:*arena*` and select `Delete edge → constants → record:commCnsts`.

`system:sys` is now defined as simply a *permanent assemblage* belonging to the *arena*.

Now define a `categorySet` to partition data between the disturbance and community. Category sets contain mutually exclusive categories: something can belong to one or the other but not both. Since disturbance and community is all there is in this model, this is the 'world' of the model so this seems a reasonable name for this set of categories.

20. Create a `categorySet` as child of `structure:struc1` and name it 'world'.
21. Create a `category` as child of `categorySet:world` and name it 'distCat'.
22. Create another `category` as child of `categorySet:world` and name it 'commCat'.

Define the data for these new categories.

23. Right-click on `category:commCat` and create the following edges:
    i. `drivers → record:commDrvs`.
    ii. `constants → record:commCnsts`.
    iii. `decorators → record:commDecs`.
24. Right-click on `category:commDist` and select `New edge → constants → record:distCnsts`.

Now define the *roles* of the new component types for the community and disturbance. Both are *permanent*, *atomic*, belongs to the *component* category and accesses data through their respective categories.

25. Right-click on `componentType:commType` and create `belongsTo` edges to `category:*permanent*`, `category:*atomic*`, `category:*component*` and `category:commCat`.
26. Right-click on `componentType:distType` and create `belongsTo` edges to `category:*permanent*`, `category:*atomic*`, `category:*component*` and `category:distCat`.

Now define the relation between disturbance and the community.

27. Right-click on `relationType:distEffectComm` and create edges:
    i. `fromCategory → distCat`.
    ii. `toCategory → commCat`.

Finally, add initialisation functions for the disturbance and community component types:

28. Create an `initFunction` as child of `componentType:commType` and name it 'InitComm'.
29. Create an `initFunction` as child of `componentType:distType` and name it 'InitDist'.
30. Delete `initFunction:Init`. This was carried over from Tutorial 3 as its no longer needed.

That's all that is required in this section. To finish up, we now need to connect various various sub-trees of the graph to each other. Foremost among these is to associate processes with the new structure.

Relations between sub-trees

Currently, `process:procCommGrowth` is applied to `category:*arena*`. We want re-apply this process to `category:commCat`.

1. Expand `dynamics:sim1` from `system:sys` and re-apply the layout.

2. Delete the `appliesTo` edge between `process:procCommGrowth` and `category:*arena*` and re-apply it by creating an `appliesTo` edge to `category:commCat`.

The task list now has two tasks: to connect both disturbance processes to either a `category` or a `relationType`.

3. Create `appliesTo` edges from both `process:procDistEffects` and `process:procDistOccurrence` to `relationType:distEffectComm`.

We have inherited a data tracker from the previous tutorial that tracks the population *x*. We need a second data tracker to follow the species diversity index 'div'. But first we need to ensure the data tracker of *x* is tracking the correct `component`. Currently, it's tracking `system:sys1` (**and not complaining!**),

1. Delete the edge from `dataTracker:trk1` to `system:sys1`.
2. Create a `trackComponent` edge from `dataTracker:trk1` to `component:comm`.
3. Create a `dataTracker` as child of `process:procCommGrowth` and select the default type `DataTrackerDO`.
4. Create a `trackField` edge from `dataTracker:trk2` to `field:div`.
5. Create a `trackComponent` edge from `dataTracker:trk2` to `component:comm`.

User interface

The specifications have inherited a controller, time series and table for *x* from the previous tutorial. We just need one more time series to view the species diversity index ('div'). We can take this opportunity to learn more about defining the simulator's user interface. The `tab` node can contain upto two widgets or two containers that in turn can contain widgets (or a combination of both). What we need then is:

a. tab
    i. table widget
    ii. container
        A. srs x widget
        B. srs diversity widget

This means we need to insert a `container` and set the `widget:srsx` and the new `widget:srsdiv` as children of it. We don't need to delete them and their edges. Instead, we can delete the parent-child link, insert a container and re-establish the parent-child links.

1. Expand the `userInterface:gui` sub-tree from the root and re-apply the layout.
2. Right-click on `tab:tab1` and select `Delete child edge → widget:srsx`.
3. Create a `container` as child of `tab:tab1`.
4. Create a new child edge from `container:cont1` to `widget:srsx`.
5. Create a new `widget` as child of `container:cont1`, name it 'srsdiv' and select its class as `SimpleTimeSeriesWIdget`.
6. Create a `trackSeries` edge from `widget:srsdiv` to `dataTracker:trk2`.

The simulation can now be run but, of course, we have yet to add code to the various functions. Here, we'll just add code snippets but if you prefer, you can create a java project and add the code there instead.

Java code

1. Add a `snippet` to each of the three `functions` and two `initFunctions` in the specifications. Add the following code to the `JavaCode` property of each snippet:

`function:InitComm`:

```java
double initFreq = 1.0 / x.size();
focalDrv.x.fillWith(initFreq);
for (int i = 0; i < r.size(0); i++) {
        focalCnt.r.setByInt(random.nextDouble(), i);
        focalCnt.K.setByInt(5.0 + initFreq + random.nextDouble(), i);
        for (int j = 0; j < alpha.size(1); j++) {
                if (i == j)
                        focalCnt.alpha.setByInt(1.0, i, j);
                else
                        focalCnt.alpha.setByInt(max(0.0001, random.nextDouble()), i, j);
        }
}
```

`function:InitDist`:

```java
focalCnt.freq = 5 + random.nextInt(50);
focalCnt.inten = random.nextDouble()*100;
```

`function:CommGrowth`:

```java
// growth
double[] dxdt = new double[x.size(0)];
for (int i = 0; i < x.size(0); i++) {
        double sum = 0;
        for (int j = 0; j < alpha.size(1); j++)
                sum += alpha.getByInt(i, j) * x.getByInt(j);
        dxdt[i] = r.getByInt(i) * x.getByInt(i) * (1 - sum / K.getByInt(i));
}
for (int i = 0; i < dxdt.length; i++)
        focalDrv.x.setByInt(Math.max(x.getByInt(i) + dxdt[i] * dt, 0.0), i);

// compute diversity
double xtot = 0.0;
for (int i = 0; i < focalDrv.x.size(0); i++)
        xtot += focalDrv.x.getByInt(i);
focalDec.div = 0.0;
for (int i = 0; i < focalDrv.x.size(0); i++)
        if (focalDrv.x.getByInt(i) > 0.0)
                focalDec.div -= (focalDrv.x.getByInt(i) / xtot) * log(focalDrv.x.getByInt(i) / xtot);
```

`function:DistOccurrence`:

```java
double proba = 1.0 / freq;
if (random.nextDouble() < proba)
        return true;
else
        return false;
```

`function:DistEffectsComm`:

```java
for (int i = 0; i < x.size(); i++)
        if (x.getByInt(i) > K.getByInt(i) * inten / 100000.0)
                otherDrv.x.setByInt(otherDrv.x.getByInt(i) * K.getByInt(i) * inten / 100000.0, i);
```

5.4.3. Next

The next tutorial introduces the event timer to drive disturbance.

Version 1.5
Last updated 2021-04-29 15:39:28 +0200