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 1.2, 10 February 2021 |

Table of Contents

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

Version: **1.2** (10 February 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 (dust to dust, ashes to ashes), are part of it and at one and the same time, they constitute an entire new world where concepts such as reproduction, social interaction, communication, life cycle, but also death, dominate the game. To this day, reconciling these two aspects 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 does 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 biologial aspects of ecosystems*. Existing tools are either (1) focusing on the physics (e.g. dynamic models based on differential equations) and can only represent reproduction and growth with metaphors for continuous physical quantities (e.g. matrix population models); or (2) focusing on the biology (e.g. multi-agent systems describing social interactions in detail but posing difficulties to balance a mass or energy budget).

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). The 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 spread 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.

Changing the scale of a model is not only a matter of space and time resolution relative to an order of magnitude. 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 detail there is in a model, the less abstract it is. At what level of detail should we model our system to capture its important behaviour? How does changing scale (in the broadest sense) effect model outcomes? There is no general answer to this question. This makes the possibility to test and compare different representations of the same system at different scales a must-have feature of any software dealing with ecosystems. **3Worlds** provides *a maximal flexibility in ecosystem representation* enabling to represent *any* ecosystem at *any* spatial and temporal scale with *any* level of detail — all this being set by the end-user in accordance with the purpose of their study. The end-user has full control on the detailed biological functions and variables relevant to their problem.

How is this compatible with the strong conceptual backbone to build required by the previous demand for easy change between scales? **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 make sure that any user-defined representation of an ecosystem complies with the archetype. This guarantees that, despite a great freedom left to the end-user, 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*, different to what usually happens in large model intercomparison exercises.

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 dream of.

How is this possible in one single software? Well, **3Worlds** builds upon more than thirty years of experience of its developers in ecosystem and complex system modelling and simulation. Over this time, we 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 modellers often have well-established solutions, sometimes in other scientific fields, (3) complex systems are not *that* complex once properly designed. Things are just available around here, it's just a matter of making them work together, and **3Worlds** is our best try 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 component, 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 latter concept means. Complex systems are usually supposed 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 coud be ideally represented with a *dynamic graph*.

According to graph theory (https://en.wikipedia.org/wiki/Graph_theory), a *graph* is a mathematical object comprising a set of *nodes*, a set of *edges* linking these nodes according to an *incidence function* (Fig. 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.

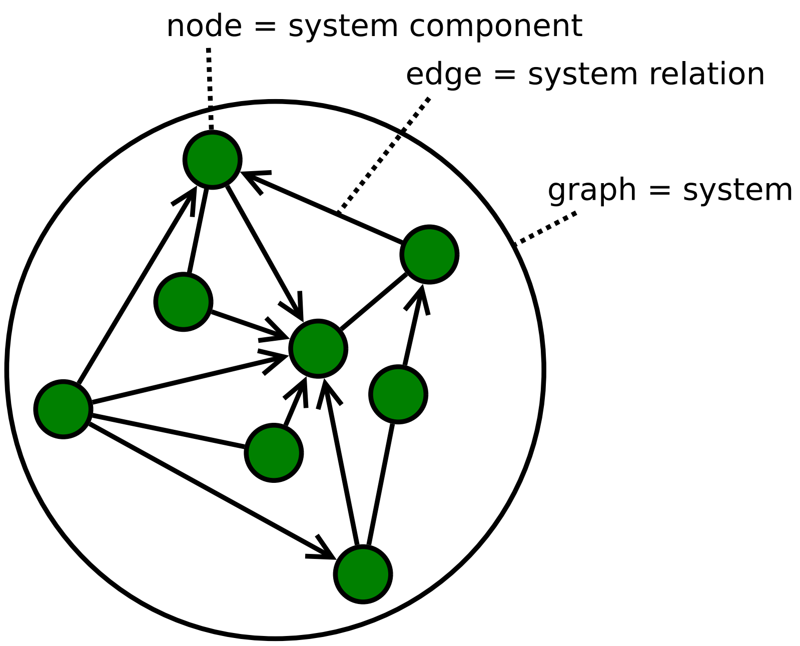
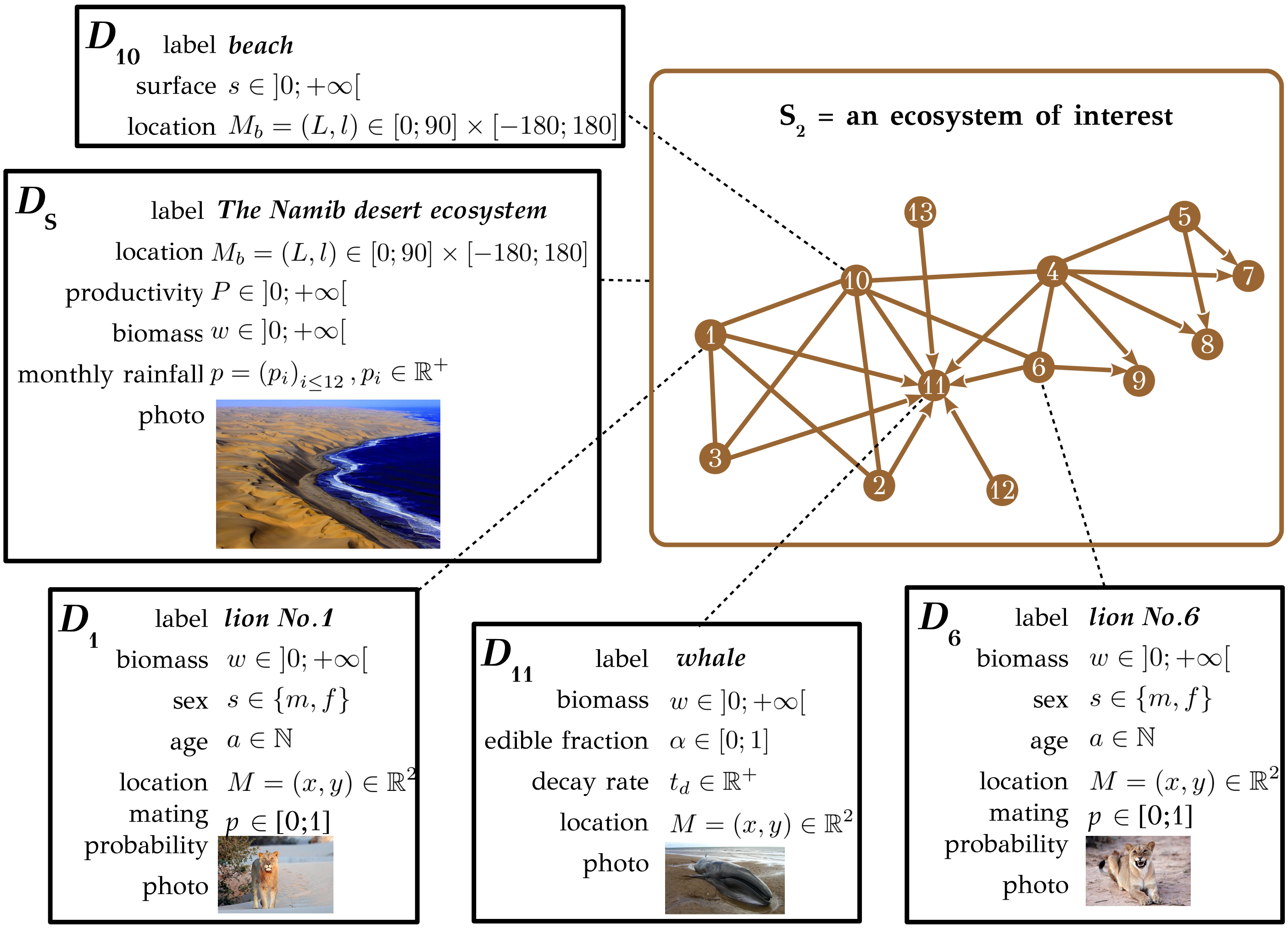


Figure 1. A mathematical graph, showing both the usual mathematical terms and the vocabulary used in 3Worlds. The incidence

function is the particular pattern of which node is related to which one. Edges can be directed (with arrows) or undirected.

If we allow to attach *descriptors* (e.g. numbers with a specific meaning: Fig. 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, we are able to make our graph dynamic. A 3Worlds simulation runs a dynamic graph where nodes, edges, descriptors, functions can be adapted to user needs for their particular application. Nodes are called *system components*, edges *system relations*, and descriptors *properties*.



D_{10} label *beach*
surface $s \in]0;+\infty[$
location $M_b = (L,l) \in [0;90] \times [-180;180]$
S_2 = an ecosystem of interest
D_S label *The Namib desert ecosystem*
location $M_b = (L,l) \in [0;90] \times [-180;180]$
productivity $P \in]0;+\infty[$
biomass $w \in]0;+\infty[$
monthly rainfall $p = (p_i)_{i \leq 12}, p_i \in \mathbb{R}^+$
photo
1
2
3
4
5
6
7
8
9
10
11
12
13
D_1 label *lion No.1*
biomass $w \in]0;+\infty[$
sex $s \in \{m,f\}$
age $a \in \mathbb{N}$
location $M = (x,y) \in \mathbb{R}^2$
mating probability $p \in [0;1]$
photo
D_{11} label *whale*
biomass $w \in]0;+\infty[$
edible fraction $\alpha \in [0;1]$
decay rate $t_d \in \mathbb{R}^+$
location $M = (x,y) \in \mathbb{R}^2$
photo
D_6 label *lion No.6*
biomass $w \in]0;+\infty[$
sex $s \in \{m,f\}$
age $a \in \mathbb{N}$
location $M = (x,y) \in \mathbb{R}^2$
mating probability $p \in [0;1]$
photo

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

1.2.3. Categories

In an IBM, every individual is different from all others. But it is common practice to assume that some groups of individuals share some common particularities. This is the essence of modelling: finding commonalities within an ocean of particular cases. We use the concetp 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.

Within the dynamic graph context, 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 component of the same category. They are also used to specify which *type of relation* is possible between components of diffent categories.

Th category concept use here is very close 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. They are central to the organisation and execution of a simulation in 3Worlds.

1.2.4. Time with simultaneous events

Ecosystems are both biological and physical systems. They do 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 (Fig. 3). This is where IBMs somewhat differ 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 rule as its default, but by using particular time models it is possible to relax this constraint.

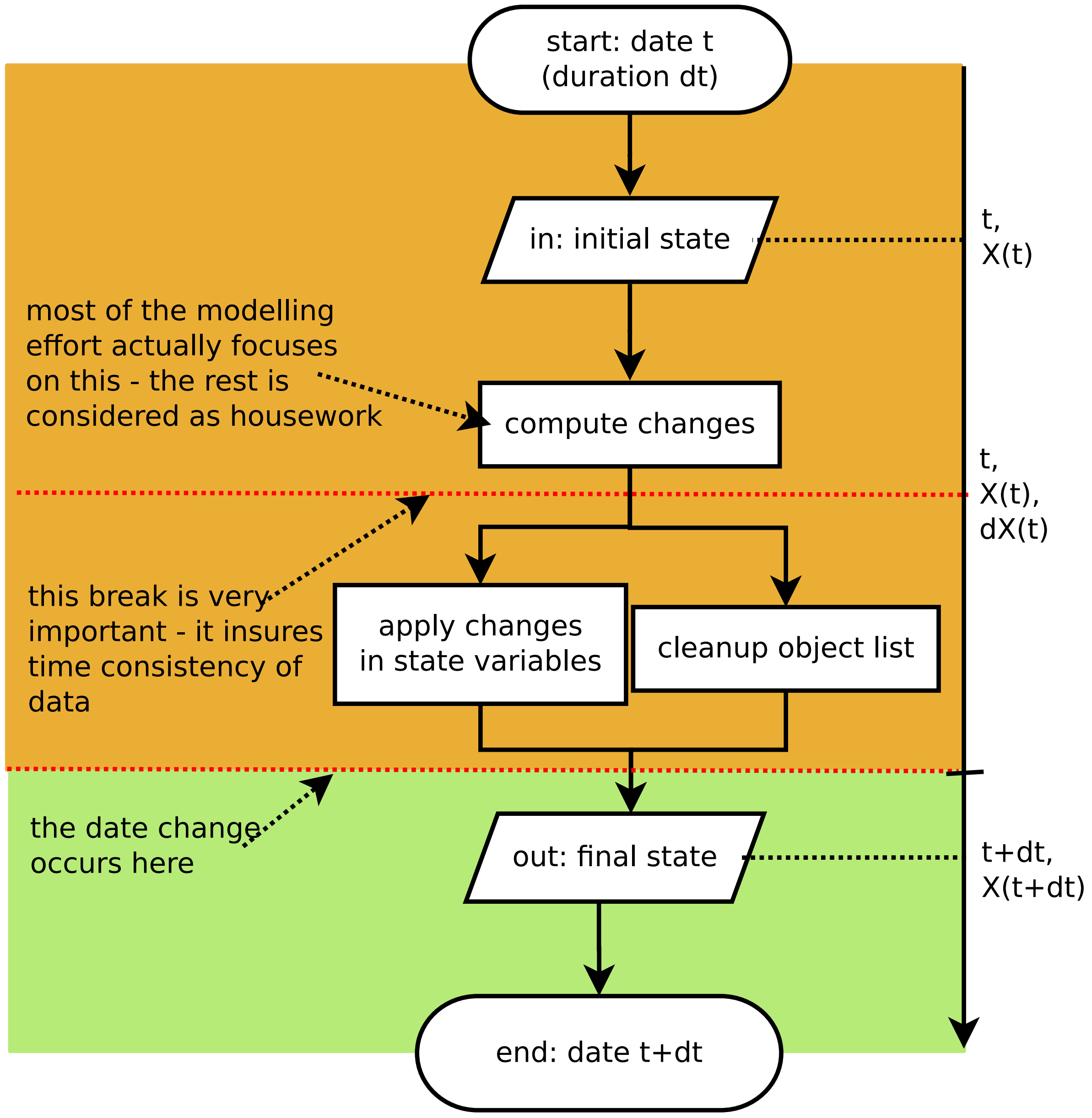

start: date t
(duration dt)
in: initial state
t,
X(t)
most of the modelling effort actually focuses on this - the rest is considered as housework
compute changes
t,
X(t),
dX(t)
this break is very important - it insures time consistency of data
apply changes in state variables
cleanup object list
the date change occurs here
out: final state
t+dt,
X(t+dt)
end: date t+dt

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. 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 where computations are to be made is user-prescribed. Time steps span the whole scale of time units possibly relevant to ecology, i.e. from milliseconds to millenia. Standard Gregorian calendar time can also be used.

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 everyone is familiar with, we called the place where organims interact in an ecosystem the *arena*, ie 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 attached to the ecosystem representation. Using a dynamic graph is sufficient.

However, it appears that in numerous cases explicitly considering space in a model helps 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 decreasing with distance distribution law; water flows in a water catchment rely 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 that can be more or less elaborate.

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 item search algorithms (e.g. Kd-trees) that speed up the search of components between which to establish relations.

1.2.6. Modelling made easy

The community of ecological scientists has been developing an impressively high number of models, yet most of them are poorly designed in terms of programming, as ecologists are not software engineers. Ecosystem simulators are among the most complex existing 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, they tend to be used beyond their initial domain of application once build (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 programing 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 end-users only have to write one code file to build a simulator for their particular problem. We used a *graph editor* to build the configuration and organise the data required for a particular use case. In 3Worlds, 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.

When designing a model, it is important to get a quick visual feedback of how the system behaves 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 user needs.

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 al yield different results. This is expected given the size of their code, but what is appalling 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 takes the form of computer codes, which are very difficult to compare above a ridiculously small code size.

3Worlds is an attempt to solve this issue *in the future* (there is nothing we can do for past model codes). 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 siulator 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 will produce ecologicallly relevant, publishable results.

IBMs are most of the time stochastic, as population rates translate into probabiities 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 random number drawings. A 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 supercalculators.

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 genericity comes at some performance cost compared to specialisation.

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 must have the following software installed on your computer prior to install 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 code development environment. You must:
  - install eclipse (https://www.eclipse.org/downloads/)
  - install the e(fx)clipse plugin (https://www.eclipse.org/efxclipse/install.html) required to use 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 does not 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.Main
```

The last is the preferred method as any errors that may arise will appear in the terminal window.

to develop your model-specific code, you will need to setup a java development environment as shown in section Setting up Java.

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

2.4.1. Setting up an eclipse *i*ntegrated *d*evelopment *e*nvironment (IDE) for 3Worlds

This assumes you have downloaded `UserCodeRunner.java` and `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 `UserCodeRunner.java` in the project:
    - Select menu `File → Import`; this opens a dialog box
    - In the dialog box, select `general > File System`
    - Click the `next` button
    - Click on the `Browse` button to select the directory where `UserCodeRunner.java` is located
    - Select the proper file in the list
    - Select `/src` as the destination location in the project
    - Click the `Finish` button

    `UserCodeRunner.java` should now appear as the unique member of a `default` package, with a compile error message attached to it.
5. Update libraries required by the project:
    - Select menu `Project → Properties`; this opens a dialog box
    - In the dialog box, select `Java Build Path`
    - Select the `Libraries` tab
    - 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

    `UserCodeRunner.java` should now have no compile errors.

2.4.2. Running ModelMaker from eclipse

ModelMaker can be run as a standalone application, or from eclipse 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.mm` and click `OK`
- If a dialog box appears warning for errors, 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 profound 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`. It requires two command line arguments (we assume that you know how to setup and run a `Run Configuration` in eclipse):

- 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
- the name of the model configuration file in this directory (e.g. *test_model_9.ugt*)

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 do not forget to keep the eclipse project content synchronized with the ModelMaker project by refreshing the eclipse project as often as necessary.

3. ModelMaker reference: creating and editing a model

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 (*cf.* 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`.

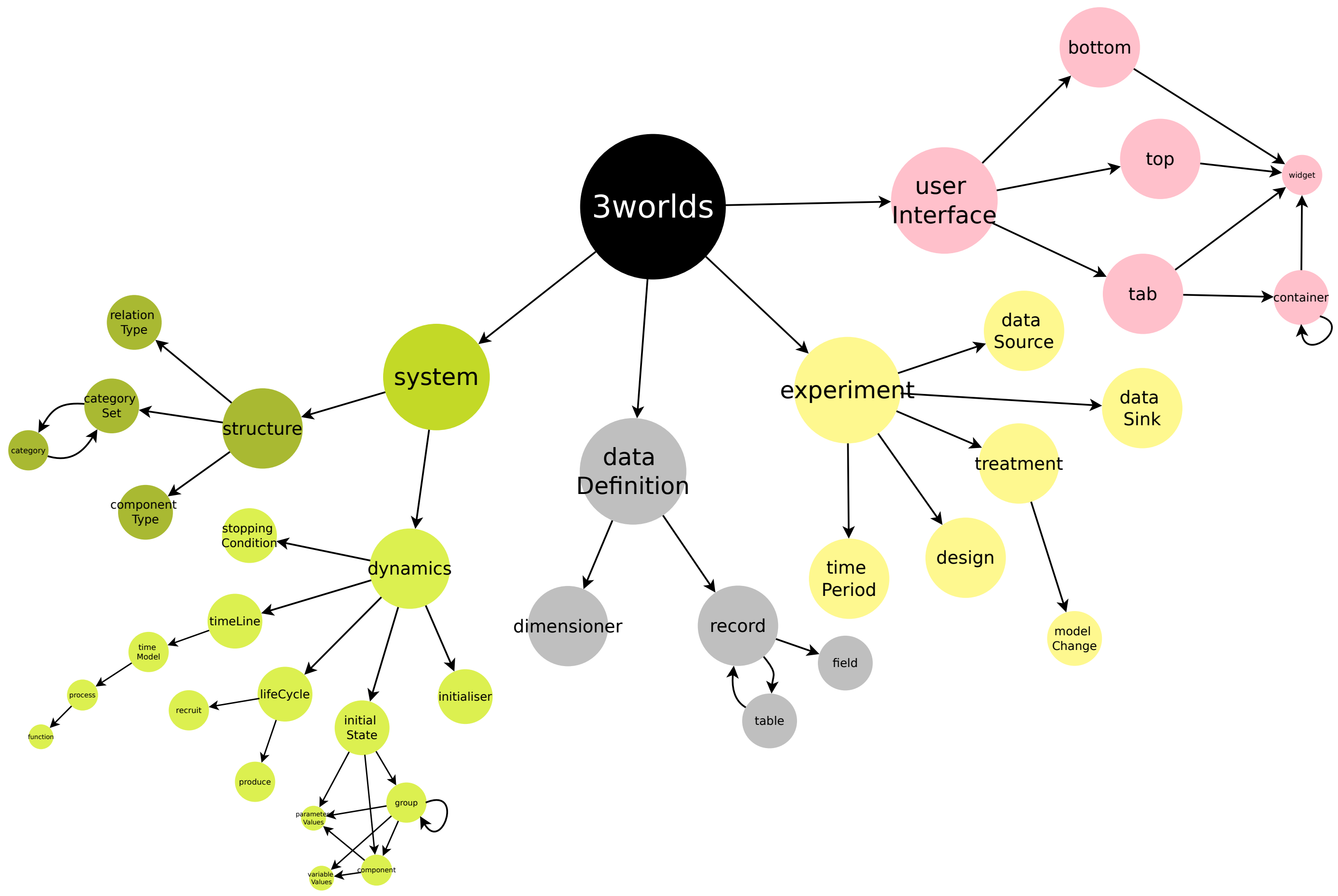


Figure 4. Tree structure of a simulation experiment configuration in 3Worlds

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).

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 3Worlds reference.

3.1.2. …with cross-links

Actually, the 3Worlds configuration is not stritcly hierarchical (*cf.* 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 fo these cross links, the whole configuration is a *graph* rather than a tree.

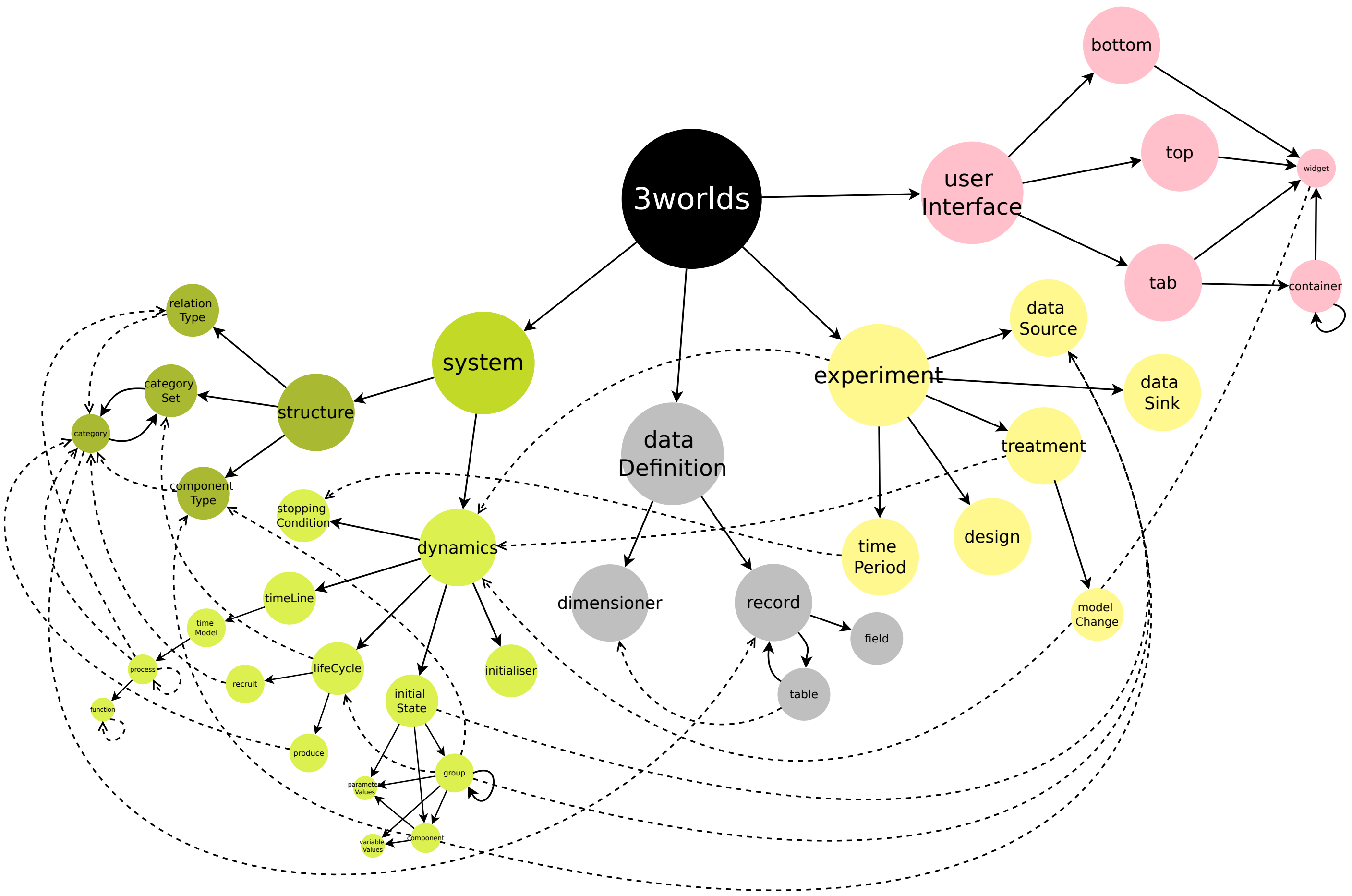


Figure 5. Tree structure of a simulation experiment configuration in 3Worlds, showing the cross-links between tree nodes

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 [Configuration options: reference].

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 `/`.

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`).

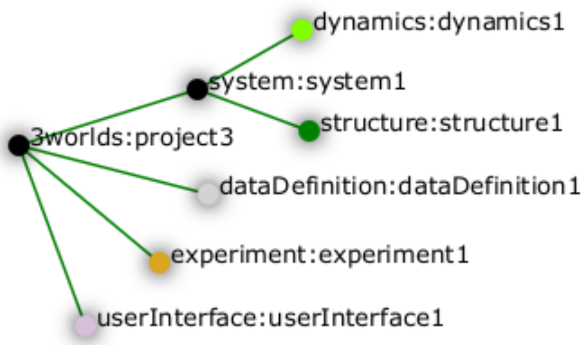


Figure 6. The base tree of any 3Worlds configuration

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`

3.3.3. The *system/structure* node

`/3worlds/system/structure:<name> {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. Practically, these common properties are *state variables*, *parameters* and dynamic behaviours (or *processes*). Categories, grouped into `categorySet`s, constitute a user-defined *classification* of *system component types* relevant for a particular model.

To make this classification useful, we attach *parameters* and *state variables* to categories. State variables are 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. Parameters do not vary during a simulation; they are characteristic of a group of system components sharing them — conceptually, a 'species' (e.g. average individual growth rate, mortality rate…).

Practically, this means that any instance of a system component of a given category will implement the state variables of this category and share parameters with other system components of the same 'species'.

The exact data structures for state variables and parameters are specified under the dataDefinition node and linked to the category through the following:

Cross-links for `category` :

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

This link tells which record data structure (in codeSource) is used to store parameters.

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

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

Similarly, these links tell which data structure in dataDefinition is used to store state variables. State variables are further classified into *drivers*, i.e. variables that *drive* the dynamics of the system; and *decorators*, i.e. secondary state variables which values are computed from those of the drivers, only reflecting the dynamics of the drivers.

A category may be defined with no parameters, drivers or decorators, but it would be pretty useless to have neither of them.

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*. Further, it may apply only to categories of a particular type (categorySet). Hence categories can be nested:

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

RelationType

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

A `relationType` is just a name representing a meaningful link between two 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. RelationTypes 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`	(1) system component stays forever during a simulation, (2) relation stays as long as both its ends stay (default value)
`ephemeral`	system component / relations are created and deleted during a simulation by the means of the appropriate `Function` classes (e.g. `DeleteDecision`, `CreateOtherDecision`, etc. cf. `TwFunctionTypes`)

Example: a category tree

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

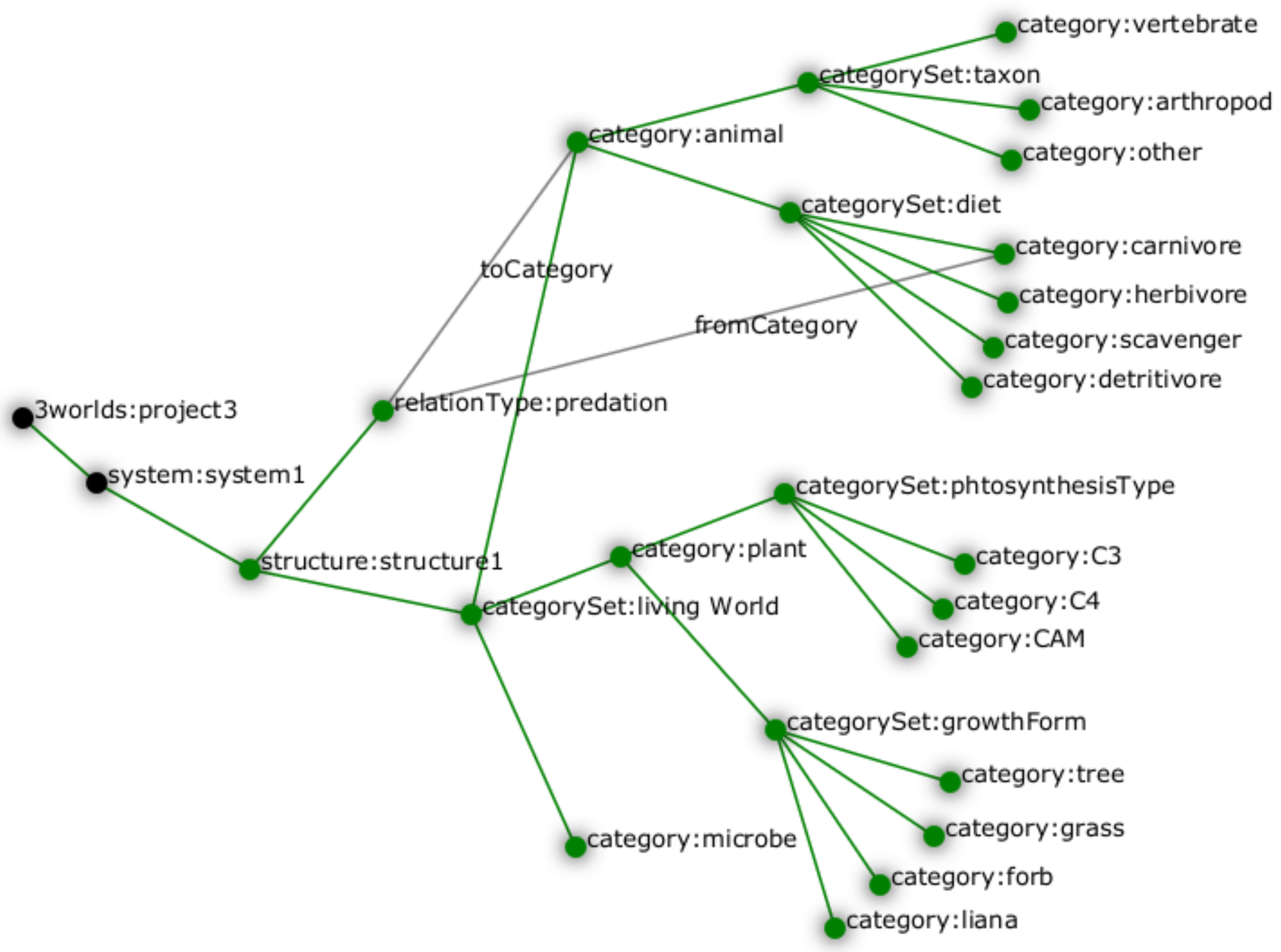


Figure 7. Example of a configuration with category sets, categories and relations

In this example, a *plant* can be a *C3 tree* but cannot be simultaneously a *grass* and a *liana*. 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).

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 state variables, 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 categories 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*).

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

Use this optional link to specify a function to initialise state variables of a `componentType` at the beginning of a simulation.

Properties for `component` :

`lifespan`	This property specifies if this type of system component will stay forever, or may get created and deleted during a simulation

possible values:

`permanent`	(1) system component stays forever during a simulation, (2) relation stays as long as both its ends stay (default value)
`ephemeral`	system component / relations are created and deleted during a simulation by the means of the appropriate `Function` classes (e.g. `DeleteDecision` , `CreateOtherDecision` , etc. cf. `TwFunctionTypes`)

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 relations. 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:

`MONO_UNIT`	single time unit, calendar-compatible (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
`ARBITRARY`	arbitrary time units with no predefined name

`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

Time models

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

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

Properties for `timeModel` :

`timeUnit`	the base time unit used by this model. *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)
`runAtTimeZero`	whether model state must be computed at time origin, *i.e.* before simulation start
`class`	the type of `timeModel` to use.

possible values:

`ClockTimeModel`	Time is incremented by a constant amount *dt*. This is commonly used to simulate regular processes like growth.
`EventTimeModel`	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.
`ScenarioTimeModel`	**Not yet implemented**.

Additional properties when `class = ClockTimeModel`

`dt` The constant time increment used in this `ClockTimeModel` , expressed as an integer number of `TimeModel` base unit (= `TimeModel.nTimeUnits x TimeModel.timeUnit`). For example, if the TimeModel 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 28 and 31 days according to the exact date).

Additional sub-tree when `class = EventTimeModel`

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

An `EventTimeModel` maintains a queue of time events that gets populated by ecological processes. Time events are stored in this queue based on their date and activated by the simulator following time order.

Cross-links for `eventQueue`

`populatedBy → function:<`*name*`>` **{1..*}**

These links indicate which ecological processes will populate the event queue with time events.

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`

`class`	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 is reached.

`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`

`endTime`	The time at which the simulation will stop, in time line `shortestTimeUnits`.

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

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

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.

To represent category changes, start and end of life, we need the concept of a *life cycle*: a series of successive discrete states decribed by different categories (starting from non-existence and finishing alike). Notice that the life cycle is optional: if a system component is going to be represented by the same set of variables (same category) during its whole life, there is no need to attach a life cycle to it as there will be no transition to other categories.

Life cycle

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

As *system component*s are designed to represent — among other things — individual organisms, they are able to create other system components ar 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. `recruit` and `produce` nodes linking categories within this set to describe possible recruitment and reproduction pathways;
3. optionnally, a `category` can be linked to a `lifeCycle` if one wants to attach parameters to the life cycle.

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

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 at least one**[maximum one? it returns a category name]** `ChangeCategoryDecision` function (cf TODO).

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 at least one `CreateOtherDecision` function (cf TODO).

Example of a life cycle specification

This life cycle:

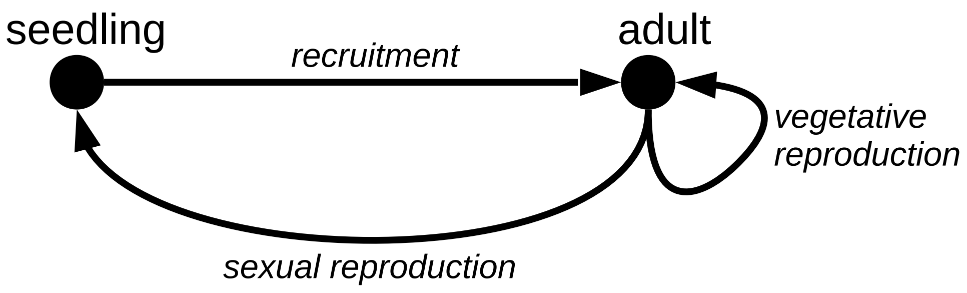


Figure 8. Example of a life cycle

is specified with this graph:

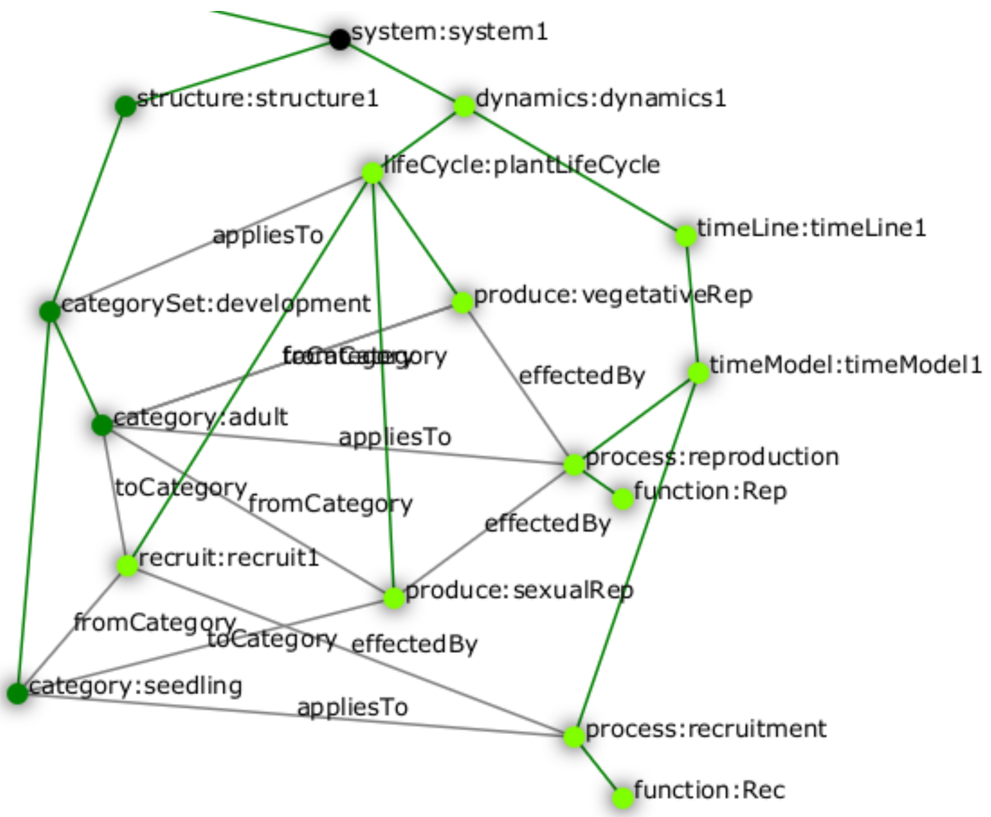


Figure 9. Example of a life cycle configuration

Ecological processes

Process

`/3worlds/system/dynamics/timeLine/timeModel/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 (cf. Category, System Component) and is scheduled by a particular time model (cf. time representation). 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 (time models). A `process` is run according to its parent `timeModel`.

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.

`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.

Function

`/3worlds/system/dynamics/timeLine/timeModel/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. This is specified by setting the `type` property to the desired function type.

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, ie the values of internal variables, 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
`Aggregator`	compute statistics
`ChangeOtherState`	*focal* changes the state of *other*
`ChangeOtherCategoryDecision`	*focal* changes the category of *other* (requires a life cycle for *other*)
`DeleteOtherDecision`	*focal* deletes *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

The selection of a function type will trigger the generation of a java source file **[WHERE?]** for a class having the name specified in the `function` node name.

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

This java source file is expected to be edited by the modeller in order to implement her/his favourite version of the ecological process modelled by the function.

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`		

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

Data tracking

[TO DO]

The setup of an initial state for a simulation

[TO DO: check the consistency of all this]

Initial state

`/3worlds/system/dynamics/initialState:<`*`name`*`>` {0..1}

To run a simulation, an initial population of `components` must be provided. Nodes under the `initialState` node are used to input data to create such an initial state. The initial state is kept in memory by the simulator, and is re-loaded at every reset prior to a new simulation of the simulator.

Group

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

Initial system components belong to groups of individual components with common characteristics. A group is either:

1. a number of components that share the same selection of categories and the same parameter set
2. or a group of previous type groups (1), belonging to the same life cycle.

Cross-links for `group`:

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

This link tells which `component` node defines the individual components of this group (type 1 above).

`cycle → lifeCycle:<`*`name`*`>` **{0..1}**

This link tells which `lifeCycle` node defines the grouping of 'groupOf'-type groups (type 2 above).

Exactly one of these cross-links must be present for every group.

Type (1) groups may be nested into type (2) groups, i.e.:

`/3worlds/…/initialState/group/group:<name>` {1..*}

Individual

`/3worlds/…/initialState/individual:<name>` {0..*}

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

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

This node specifies an instance of a *system component* to be created at the beginning of a simulation.

Cross-links for `individual`:

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

This link tells which `component` node defines this individual component.

Parameters

Parameter sets define a group: they are constant values which are shared by all individual members of a group.

`/3worlds/…/initialState/parameterValues:<name>` {0..*}

Use this node to attach a set of parameters to the whole ecosystem. There will be only one of such parameter sets in any given simulator. In user-defined functions, these parameters are available through the `focalContext.ecosystemParameters` field.

all focalContext fields may be null if no value was provided at initialisation. Any user-edited function code must check for this possibility when using these fields.

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

Use this node to attach a set of parameters to a group. Each different *name* will create a different parameter set. Depending on the group type, these parameters are available through `focalContext.lifeCycleParameters` of `focalContext.groupParameters` in the user-defined functions.

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

Use this node to attach a set of parameters to an individual system component.

this implies that this system component is unique of its kind, and no other component is allowed to share these parameter values.

[TODO: check this is not a flaw - how do we access these parameters ?]

Variables

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

Use this node to attach a set of initial driver values to an individual system component. A new component will be created for every different value of *name*.

[TO DO: loading from files...]

Old version of documentation after this line - dont read.

3.3.5. The *codeSource* node

`/3worlds/codeSource` {1..*}

This node and its sub-tree contains specifications for automatic code generation needed to implement a particular model. Most of its nodes have cross-references to nodes of the `ecology` sub-tree. The multiplicity allows users to organise their code specifications into meaningful units

3.3.6. The *dataIO* node

`/3worlds/dataIO` {1..*}

This node and its sub-tree contains links to external data sources (usually files) required by a simulation experiment, either for data input or output. Most of its nodes have cross-references to nodes of the `ecology` sub-tree. The multiplicity allows users to organise their data sets into meaningful units.

3.3.7. The *experiment* node

This part is still under construction

`/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. The name is used to differentiate simulation experiments in a meaningful way. **TODO[It will appear in output directory names ?]**.

The `experiment` node must have a cross-reference edge labelled `baseLine` {1} to an `ecology` node. The model configuration contained in this `ecology` sub-tree will be used as the reference, "base line" simulation in the experiment - similar to a control treatment in a real-world experiment.

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

3.3.8. 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, for example as help when debugging a new model; or only show a progress bar to improve computing performance when running a big simulation experiment.

3.3.9. The *hardware* node

This part is still under construction. The default settings should be used.

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

This node and its sub-tree specifies how the experiment should be distributed on available harware. **At the time of writing, only deployment to a single computer can be done**.

3.4. Developing and testing model code

This section gives general rules to follow to successfully edit the generated `TwFunction` -descendant classes in order for your model to behave as you wish. Section Setting up a user coding environment describes how to link your ModelMaker session with an eclipse development enviromnent where you can edit the generated code. Section Function describes all the options you can setup in `ModelMaker` to specify your function.

All the generated function classes look like this:

JAVA

```
package system1;

import au.edu.anu.rscs.aot.collections.tables.*;
import au.edu.anu.twcore.ecosystem.runtime.system.SystemComponent;
import au.edu.anu.twcore.ecosystem.runtime.biology.ChangeStateFunction;

/*
[... some big comment...]
* Class Function1
* Model "system1" -  - Mon Sep 23 09:05:33 CEST 2019
* CAUTION: Edit this template but do not change class declaration.
*/

public class Function1 extends ChangeStateFunction {

	@Override
	public void changeState(double t, double dt, SystemComponent focal) {
		// INSERT YOUR CODE HERE
	}

}
```

In this example, "**system1**" was the name given to the `system` node, while "**Function1**" was the name given to the `function` node. The `type` property set for the `function` was "**ChangeState**", as shown by the generated code. Other function types would have caused the generation of different methods in the class.

3.4.1. generic method arguments

All generated methods share the following arguments:

- the first and second argument are *the current time **t*** and *current time step **dt***, passed by the simulator as double values in units of the `timeModel` of the parent `process` of the `function`.
- For all function types which process applies to categories (`appliesTo → category`), i.e. `ChangeCategoryDecision`, `ChangeState`, `DeleteDecision`, `CreateOtherDecision`, there is an additional argument, the system component which will be affected by the computations, called *focal*.
- For all function types which process applies to relations (`appliesTo → relationType`), i.e. `ChangeOtherCategoryDecision`, `ChangeOtherState`, `DeleteOtherDecision`, `ChangeRelationState`, `MaintainRelationDecision`, and for `RelateToDecision`, there is another additional argument, a second system component, the one at the other end of the relation starting from *focal*, called *other*.

3.4.2. generic method fields

In addition to these arguments, every function class has a protected field called `localContext` that contain contextual data that may be used in computations. The local context data consist in:

- *ecosystem*-level data, i.e. ecosystem population data (= number [source,]

```
focalContext.ecosystemPopulationData
focalContext.ecosystemName
```

- *life cycle*-level data, i.e. life cycle parameters, population data, and name:

```
focalContext.lifeCycleParameters
focalContext.lifeCyclePopulationData
focalContext.lifeCycleName
```

- *group*-level data, i.e. group parameters, population data, and name. A group represents here components sharing the same categories and same parameter set (eg *species* characteristics for living organisms).

```
focalContext.groupParameters
focalContext.groupPopulationData
focalContext.groupCycleName
```

`PopulationData` has 3 accessible fields: `count`, `nremoved` and `nAdded`.

TODO update this

- The `changeState(…)` method of the `ChangeState` and `ChangeRelationState` function types, and the `changeOtherState(…)` method of the `ChangeOtherState` function type, are expected to compute changes in state variables of the *focal* component or relation **[NB: this is not yet implemented for relations]**, or the *other* component in the case of the `ChangeOtherState` function. The state variables are found in `focal.currentState()`, a read-only set of values, and new values may be computed into `focal.nextState()`.
- The `delete(…)` method of the `DeleteDecision` and `DeleteOtherDecision` must return a `boolean` value. If `true` is returned, this will trigger the removal of the *focal* (for `DeleteDecision`) or *other* component (for `DeleteOtherDecision`).

> In all `…Decision` functions except `CreateOtherDecision`, a helper method called `decide(double proba)` is available. This method will return true with probability `proba` and can be used to return a boolean result based on a probability computation. This method uses the in-built random stream facility of 3Worlds.

- The `nNew(…)` method of the `CreateOtherDecision` function takes an extra argument called `newType`, which is the category name of the newly created component, as per the life cycle. `nNew(…)` returns a number of new components to create as a decimal number (`double`): the *integral part* directly translates into a number of new components, while the *decimal part* is used as a probability of an extra new component. This way, very low fecundity probabilities can be simulated.

> If there are >1 possible descendant categories, the `nNew(…)` method will be called as many times, with the `newType` parameter changing accordingly. The `nNew(…)` method must be prepared to handle multiple choices in such case (for example with a `switch` statement).

- The `changeCategory(…)` method of the `ChangeCategoryDecision` and `ChangeOtherCategoryDecision` function types returns a category name (`String`), that of the new recruit. This name must be consistent with the life cycle information. After the function is executed, the *focal* (for `ChangeCategoryDecision`) or the *other* (for `ChangeOtherCategoryDecision`) component is recruited to its new category.
- The `maintainRelation(…)` method of the `MaintainRelationDecision` and the `relate(…)` method of the `RelateToDecision` both return a `boolean` value. If `true` is returned, the relation is maintained/set between the two components, otherwise it is removed/not set.**[TODO: a lot to add when indexers come in]**

3.5. Feeding the model with data

4. ModelRunner reference: running a simulation experiment

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

Version 1.2
Last updated 2021-02-10 17:46:41 +0100