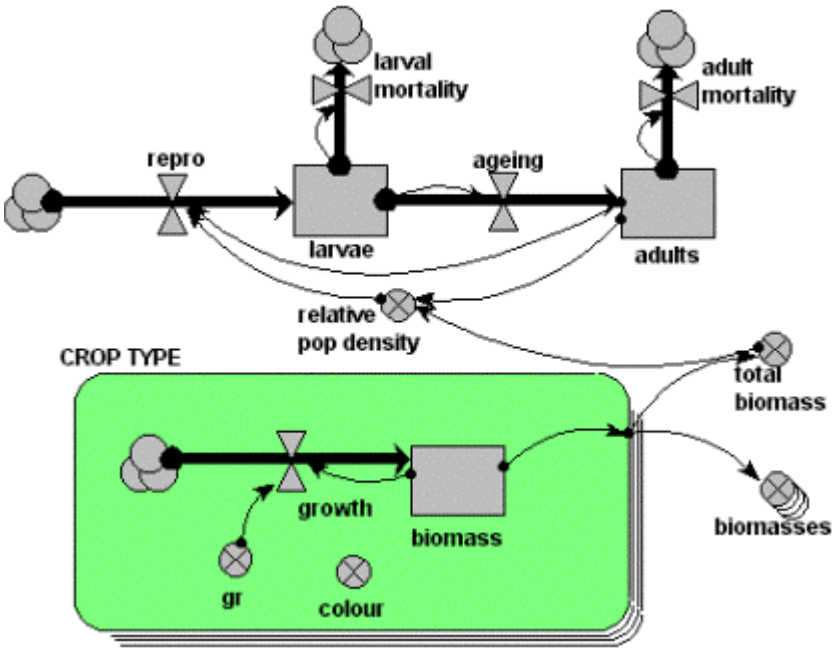
Getting to know

Simile

the visual modelling environment for ecological, biological and environmental research



Robert Muetzelfeldt
Jasper Taylor

The University of Edinburgh
Institute of Ecology and Resource Management

http://www.ierm.ed.ac.uk/simile
r.muetzelfeldt@ed.ac.uk
jasper.taylor@ed.ac.uk

Getting to know Simile

the visual modelling environment for ecological, biological and environmental research

Contents

Why did we develop Simile? - problems with the current practice of ecological modelling

Simulation modelling is an important aspect of research in the ecological, environmental and related areas. Such models can guide the research process, integrate the knowledge coming from a variety of studies, permit the testing of hypotheses, and enable predictions to be made. Many major research programmes have the development of such models as a central component of the whole programme.

Most models are implemented in conventional programming languages, such as Fortran and C: some even in Basic. This greatly reduces the effectiveness of the modelling effort, for a number of reasons:

1. It requires either that the researcher learns a programming language (which they may be unwilling or unable to do), or that a programmer is employed to implement the model (which is a significant cost and adds a barrier between the model as conceived and the model as implemented).
2. Developing a reliable programmed implementation is a time-consuming hence costly operation, not only in the initial implementation of the model but also in its subsequent debugging and maintenance.
3. It is difficult for others to fully understand the model, since they will be reluctant to wade through many lines of computer code.
4. The written description of the model is often incomplete, sometimes ambiguous, and it has no guaranteed correspondence with the model as implemented.
5. Shareability of model components with other researchers is severely limited, thus leading to considerable duplication of effort. This applies both to submodels and to support routines, such as data input facilities and tools for visualising model behaviour.
6. Implementing model in a procedural programming language ignores the fact that running simulations is just one of the things we can do with a model: there are many other activities associated with the modelling process (such as comparing two models, generating descriptions of a model in a variety of formats, and transforming models) that could in principle be aided by a computer, but not if the model is implemented as a program.

These problems have been recognised for a good number of years in the modelling community, and various solutions have been proposed. First, there has been some development of modular modelling environments (such as APSIM), but these tend to make rigid assumptions about the way that submodels should integrate, and thus can at best only work within a small, well-defined community. Second, there are now a number of simulation languages (such as ACSL), and visual modelling environments (such as Stella, ModelMaker and PowerSim). These certainly reduce the programming overhead, but tend to restrict the modeller to a certain modelling approach (such as differential equations or System Dynamics modelling). Third, there is now considerable interest in the adoption of object-oriented or component-based technologies such as C++ or COM. These have the potential to permit re-use of model components between groups, but are still tied into a programming paradigm for the implementation of the individual model components.

So currently there is no clear-cut adoption of one approach or the other within the research community. This is partly because of the technical limitations briefly touched on above. However, it is also the case that the there has been virtually no attempt by the research community to adopt or devise a common modelling methodology. Most groups still do their own thing: indeed, there is in some quarters a firmly-held belief that choosing a technology for implementing a model is part of scientific freedom, and that any attempt to look at the development of common standards threatens this freedom. In our view, it is beyond dispute that a researcher who wishes to develop a model that cannot be implemented in a common framework should have the freedom to do that. However, we certainly do dispute the right of

a researcher to waste taxpayers' money implementing a model inefficiently, without re-using existing components, and developing yet more visualisation tools for displaying model behaviour.

Simile aims to demonstrate that it is possible to reduce the problems associated with the current approaches to modelling, while imposing few restrictions on the types or size of models that can be constructed. Currently, it has made considerable progress in achieving these aims, and it is now a mature product that has been used in two international research projects, and by a number of research groups. However, our overarching goal is not to get everybody using Simile. Rather, it is to catalyse discussion within the research community on the need for standard approaches to modelling, the nature of such standards, and the institutional arrangements and funding needed to bring them about. So, Simile is a 'proof-of-concept': we show some possibilities, and we show what it is like to adopt certain solutions. It can be placed on the table along with other candidate technologies.

The vision - how the research community could be doing its modelling

Imagine if....
.... you could build a model without having to learn programming, and with no restrictions on what you could put into the model;
.... models could be built in a fraction of the time currently required;
.... you could run the same model on various platforms, even parallel computers;
.... models could be seamlessly integrated with GIS;
.... you could customise the types of displays used for visualising model behaviour;
.... you could automatically generate a description for any model in a variety of formats and level of detail;
.... you could automatically compare the structure of two models, to see their similarities and differences;
.... you could share models with other people as easily as you can now share Word documents;
.... you could search the internet for models just like you can now search for web pages;
.... you could edit or run any model you find in your web browser;
.... you could download any model you wanted by clicking on a link in a web page;
.... you could query the structure of a model, just like you can query a database;
.... you could insert any model as a submodel in some larger model;
.... you could extract a submodel from a larger model, and run it as a stand-alone model;
.... you could link model variables to data sets using metadata;
.... you could attach hyperlinks to any part of a model, taking a user to pages anywhere on the internet describing its biological or mathematical basis.

You can't do all of these things with Simile - or any other modelling environment - yet. But these goals are certainly achievable, given sufficient commitment by the research community in the development of standards and software tools.

Simile and declarative modelling

Simile is billed as a 'visual modelling environment', meaning that models are developed diagrammatically (as opposed to writing lines of text, as in a programming language or a simulation language). However, a more fundamental feature of Simile is that it is a declarative modelling environment. In a sense, Simile's visual modelling interface is just one way of building a Simile model: we could envisage other interfaces (such as a text editor) that would still produce a Simile model, but would not be 'visual'.

What do we mean by 'declarative modelling'? First, we note that the term contrasts with a procedural approach, which is what we do when we implement a model in a conventional (procedural) programming language, such as Fortran or C. In that case the lines in the program are in fact instructions to the computer: calculate this value, loop over the array elements, print the value for this variable. Some of the lines might look exactly like declaratively-expressed equations, but they are in fact assignment statements. Thus, without the use of strictly-observed (and arbitrary) conventions, we cannot really distinguish between the assignment statements in the program that constitute the model equations, and all the other assignments we might have in the program. Effectively, we can do only two things with the program: run it (to calculate the values for the model variables); or show it to a human who understands the programming language. And running it is the only thing that a computer can do for us.

In declarative modelling, we represent a model not as a series of assignment and control statements, but as a set of facts that are true about the model. The order in which we present the facts is (unlike a procedural program) irrelevant. The full set of facts defining a model actually constitute a specification for the model: given these facts, and knowledge about what the symbols mean, someone else can construct a working version of the models.

A good example is in architectural design. We would never dream of specifying a design for a building in a computer program: rather, we would express it as a set of facts, defining the walls, doors etc, and their size, location and materials. Given this information, we could draw a plan for the building. We also have all the information we need to do calculations about the building: how big it is; how quickly it would heat up and cool down. The program that does the calculations is indeed procedural; but the design of a particular building is held as a set of facts.

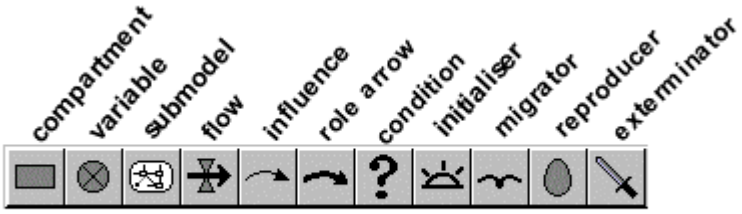
And so it can be with models. A model is a certainly a design, expressed in terms of the constituent parts of the model and the linkages between them. It can certainly be expressed declaratively: look at he 'Model description' part of any modelling paper. The declarative statement of model structure is sufficient to enable someone else to make a runnable version of the model, provided that it is complete, unambiguous, and that the meaning of the terms used is understood.

There are significant advantages in adopting an declarative approach. First, there is no risk of the description of the model failing to match the implementation of the model: the description is the implementation. Second, once a model is represented declaratively, one can do many things with it as well as just simulating its behaviour: for example, generate descriptions in a variety of formats, interrogate its structure, compare it structure with that of another model, or transform it into a simpler or more complex model. Third, one can generate runnable versions of the model in a variety of languages (including languages for parallel computers): you are not limited to one that the model happens to be written in. Finally, the adoption of a declarative modelling approach encourages the development of common standards for representing models, the distributed development of modelling tools, and the sustainability of the effort put into developing models.

Simile, in common with other modelling environments, necessarily adopts a declarative approach: it is hard to build up a representation of the model during the design process without doing this. However, in developing Simile we have also committed ourselves to the idea that the language for representing the model structure should be open, so that any other person or group can develop their own tools for processing Simile models - including tools for generating runnable versions of a model or indeed model-design environments.

The Simile modelling language

This section lists the model-design elements that you can use to construct a model in Simile. These are described in terms of their graphical representation: i.e. the symbols you place on the model diagram as you build up the model design. However, you should appreciate that the diagram is really just a way of visualising the internal state of the model specification, there being a one-to-one correspondence between a graphical symbol and a statement in the internal model specification. In principle, one could build a model purely in textual terms, by entering the set of statements defining a model directly, rather than going through the diagrammatic interface.



Compartment

We can think of a compartment as representing the amount of some substance, hence the use of other terms in the System Dynamics community, such as 'stock' or 'level'. Mathematically, a compartment represents a state variable whose behaviour is defined by a differential equation. A compartment requires an initial value, which is usually a numeric constant, but can be calculated from other constants.

Flow arrow

A flow represents a process causing an increase or decrease in the amount of substance in a compartment. It is thus represented visually by an arrow pointing into or out of a compartment, possibly connecting two compartments. Mathematically, a flow is an additive term in a differential equation for the associated compartment (state variable). The value assigned to a flow can be a constant or a function of other quantities in the model.

Variable

A variable represents any quantity whose value is either a constant or calculated as a function of other quantities in the model. In modelling terms, a Simile 'variable' can thus represent a parameter, an intermediate variable, an output variable, or an exogenous variable.

Influence arrow

An influence arrow represents the fact that one quantity is used to calculate another.

Submodel

In Simile, a submodel is a round-cornered box that encloses a number of model-design symbols, including possibly other submodels. In modelling terms, the single concept has a range of interpretations, ranging from a purely visual dividing up of a complex model into its component parts (e.g. 'the soil water submodel'), through the defining of objects in the model (such as individual trees or animals), to defining an association between objects (such as an

ownership association between farmers and fields). Because the submodel concept is so central to Simile, has so many uses, and sets it apart from more standard approaches, the next section considers submodels in more details.

Condition

A condition element is placed inside a submodel to indicate that the existence of the submodel depends on some condition. For example, we might have a model with multiple patches of land, of which only some have a crop. In this case, we would have a crop submodel inside the (multiple-instance) patch submodel, with the crop submodel being conditional on the landuse for this patch being of type 'crop'.

Role arrow

Role arrows are usually used in pairs, with each arrow going from a submodel to a submodel. The combination of submodels and role arrows is used to denote an association between the objects represented by the submodels at the start of each role arrow. The term 'association' refers to a relationship such as the neighbour relationship between two patches of land, or an ownership relationship between farmers and fields.

Initial-number

The initial-number element is used to specify the initial number of members in a population of objects.

Migrator

The migrator element is used to specify the rate at which new members of a population are created.

Reproducer

The reproducer element is used to specify the rate at which each member of a population creates new members.

Exterminator

The mortality element is used to specify the rule for killing off a member of a population of objects.

The above set of model-design elements reflects the choices made by the Simile design team. In some cases (such as the System Dynamics elements of compartment and flow) there was in fact little choice, given the wide-spread acceptance of this diagrammatic notation in the research community. In other cases, we have invented elements that we saw as being required to meet a certain modelling need.

This set is almost certainly minimal: it is hard to see how the number of elements could be reduced further, without making it impossible to express certain modelling concepts. Conversely, it is very easy to propose new elements which are not strictly necessary (i.e. their function could be replicated by some combination of existing elements) but which might simplify the model diagram or enhance its readability. One example is the introduction of a 'memory' element, which stores a value until it gets a signal to change it. This can be done

using a compartment-flow combination or a pair of variables linked by the 'last' function, but neither of these is a clean way of diagramming the basic concept. The problem is that, once you start going down this route, then there is the risk of a rapid explosion in the number of symbols available to the modeller, and one has to trade off the advantages of having more symbols against the increased learning required to read a diagram.

We do not claim that the set of model elements that we have come up with in Simile is optimal. However, in designing Simile, we have become very aware of the issues involved in designing the elements for a modelling language, and of the trade-offs involved between having a small number of more abstract elements versus a larger number of more concrete elements. These issues and trade-offs need to be given careful consideration in any attempt to design a generic modelling environment.

The various uses of the Simile 'submodel'

A submodel in Simile is a way of bundling up a number of model elements, including other submodels. This is done by either drawing a submodel envelope around a number of elements in the model diagram, or by creating an empty submodel and inserting model elements into it.

However, the reasons for wanting to do this are many and varied, and it is important to appreciate that the submodel construct can be used for a range of modelling needs. These are normally considered as being pretty separate, so it may come as some surprise to see that the same model-diagram construct can be used for these different purposes. However, there are considerable benefits by using a single method, both in terms of what you need to learn, and in terms of the machinery that lies inside Simile.

This section overviews the different uses of the submodel construct, and the different types of submodel that you can have.

Using a submodel to move a block of model diagram elements

Simile provides a Move tool, but you can (currently) move only individual model-diagram elements: there is no ability to select multiple elements simultaneously. But what you can do is to draw a submodel box around them, shift the whole submodel (by dragging anywhere inside it), then deleting the submodel boundary. Not the primary reason for having a submodel concept! - but a useful diagram-editing technique to know about.

Using a submodel to show the main components of a complex model.

You have constructed a model with a number of compartments and flows. Some relate to vegetation; some to the animals in the area; some to soil water and nutrients. By grouping the model-diagram elements for these different parts into submodels (called 'Vegetation', 'Animals' and 'Soil'), the gross structure of the model is immediately apparent.

Conversely, you may prefer to design a model in a top-down fashion. Starting with a blank screen, you can rapidly add submodels corresponding to the main components of a proposed model, then subsequently add the various compartments, flows etc inside these.
Using a submodel to move parts of a model diagram from one place to another.
You can move individual model elements using the Move tool in the toolbar, but how do you move a set of elements? All you need to do is to draw a submodel envelope around them, move the whole submodel to another area of the model diagram, then delete the submodel boundary.

Using a submodel for multiple views on a model, perhaps at different scales.

Once part of a model is made into a submodel, you can open up a separate window for it (by double-clicking on its boundary). This window can be kept on the screen while you scroll the main model diagram to some other part of the model. Also, you can change the zoom factor for each main model window or submodel window separately, enabling you to see part of the model in fine detail while maintaining an overview of the whole model at a coarser scale.

Using a submodel to make a part of a model into a stand-alone model.

For the model described above, you may want to see how the vegetation part behaves, assuming fixed inputs from the animal and soil sections that affect it. You draw a submodel envelope around the vegetation, open up a separate window for it, then use the File: Save command to save it to a file. You can then start up Simile again, and load just the saved vegetation submodel (which is now a model in its own right). You can now explore how it behaves by itself. This can be very useful for testing and debugging purposes.

Using a submodel for modular modelling: swapping one module for another.

For many years, the battle cry of those fed up with the implementation of models in computer programs was 'modular modelling'. If we had a modular modelling system, it was argued, then models could be easily constructed from a number of pre-programmed modules, and the effectiveness of the community as a whole would be greatly increased by the sharing of these modules, avoiding huge duplications of effort.

The submodel concept in Simile supports modular modelling. You can open up a separate window for a submodel (say, a vegetation submodel); clear the contents of the submodel (by doing File: New), then load a different vegetation model into the submodel window. Influence links with the rest of the model can then be made one by one.
Furthermore, Simile supports 'plug-and-play' modularity (which is what is normally meant by 'modular modelling'). If two or more vegetation submodels have been designed to share a common set of influences (in and out) with the rest of the model, then Simile the information about this interfacing to be stored in a file (an interface spec file). When you next load one of the submodels from a file, you simply refer to the interface spec file, and all the influence links are made in one quick operation.

Using a submodel for disaggregation; or (conversely) specifying a fixed number of objects of a certain class.

These two terms are lumped together because they are the same concept, seen from opposite perspectives. You can disaggregate an area into a number of patches; or you can think in terms of one patch, then have multiple patches to represent some larger area. The end result in both cases is exactly the same.

Once you have made a submodel you can specify (by going to its Properties box) that it is a 'fixed-membership submodel', and specify a number of instances. The submodel then represents each of that number of instances. Visually, it now appears different, because it now has multiple lines on the left- and bottom-edges: like a stack of cards. Internally, Simile now handles each instance separately: each can have its own parameter and initial values, while they all have the same compartments, flows etc.

This enables many forms of disaggregation to be captured. For example:
- disaggregating a population into age, size, or sex classes;
- disaggregating a vegetation component into the several species that make it up;

- disaggregating soil or forest canopy into a number of layers;
- disaggregating space into grid squares, polygons, or some other form of spatial unit.

Using a submodel to specify a dynamically-varying population of objects.

The modelling world divides into those whose models are based on differential/difference equations (with or without disaggregation); and those who subscribe to an approach based on collections of objects (variously called object-oriented, individual-based or agent-based modelling). The latter group is much smaller, but evangelical and rapidly increasing in size.

Simile enables a population approach to be combined with a differential-difference equation approach. For example, a modeller might represent the vegetation in terms of compartments and flows, while the herbivores might be represented as individual animals, which are created, grow and die. In order to do this, a submodel is specified as being a population submodel (again, in its Properties box), and model elements can be added for specifying the initial number, and the rules for the creation of new individuals and the elimination of those already n the population. Visually, the submodel now appears with a shadow line for the top- and -left edges, and another for the bottom- and right-edges.

Using a submodel to specify the conditional existence of some part of the model.

When a model is implemented in a conventional programming language, large chunks of the program can be enclosed inside an if…endif block: i.e. whether it is actually evaluated depends on some condition. This programming device may be applied to several different purposes:

- You may want to have several alternative ways of modelling some part of the system (e.g. a growth function), only one of which is active in any one run of the model. A flag determines which one is active.
- You may want to model a set of species using a single submodel, but with only some species present in any one run of the model.
- You may want to model a number of spatial patches, some of which contain one land use type, and others of which contain another. You need to include a submodel for each one within the multiple-instance patch submodel - but switch one or the other on in a particular patch.

All these situations can be handled in Simile using a conditional submodel. This is simply a normal submodel, but with a condition symbol added. Visually, we can tell that it's a conditional submodel both by the presence of the condition symbol, and by a set of dots going down diagonally to the right from the submodel envelope. The condition contains a boolean expression: if this evaluates to 'true', then the submodel (or an instance of it) exists; if not, then it doesn't.

A conditional submodel will, like any other, have influences coming out from the model elements it contains. However, the number of values passed along each influence will either be zero (if the submodel does not exist), or one, if it does. This is thus a variable-size data structure: in other words, a list (with the name of the variable enclosed in curly braces {…}). In Simile, the only thing that can be done with a list is to evaluate it: usually, to sum its values. If the list is empty, then the sum is zero. If the list contains a single element, then the sum is whatever this value is.

Using a submodel to specify an association between objects.

Once our modelling language allows us to think in terms of multiple objects of a certain type, then it is frequently the case that we start to recognise relationships between objects. These relationships may be:

- between objects of the same type: one tree shades another; one grid square is next to another; one person is married to another; or

- between objects of one type and objects of another: one farmer owns a field; one field is close to a village.

Since Simile is a visual modelling language, and since such relationships are an important aspect of the design of a particular model, Simile provides visual elements to show diagramatically such relationships between objects. Unfortunately, the term 'relationship' is normally used in ecological modelling to refer to a relationship between variables (as opposed to objects), so we use the term 'association' instead. This is the same term used in UML (the Unified Modelling Language, the standard object-oriented design language used in the software-engineering community).

An association can itself have properties. We can, for example, have a variable representing the actual distance between a field and a village: this is a property of neither the field or the village, but of the association between them. In Simile, the submodel is the construct that is able to hold a number of quantities, therefore we use a submodel to represent an association: it is then called an association submodel.
However, such a submodel is simply a normal Simile submodel. It becomes an association submodel by virtue of being linked to the submodel (or submodels) representing the objects that have the association. The linking is done using role arrows: one role arrow is drawn for each type of object that participates in the association. Thus:

- for the owns association between farmer and field, we draw a single role arrow from the farmer submodel to the owns association submodel, and one from the field submodel to the owns association submodel;
- for the next to association between one grid square and another, we draw two role arrows from the grid square submodel to the next to association submodel: one role arrow represents the field under consideration, while the other represents its neighbour.

Using a submodel to specify a 'satellite' relationship

When you pass information out of fixed-membership multiple-instance submodel, it appears as an array with a fixed number of elements. You can extract the value for any one element using the 'element([array],index)' function, and the element you extract will correspond to the instance in the fixed-membership submodel. This makes it possible to select values for one variable on the basis of the value for some other variable.

However, when you pass information out of a population submodel or an association submodel, all you get is a list of values: each value is <u>not</u> tagged with the index (instance number) of the submodel instance that produced it. Thus, if you had a population submodel for a population of rabbits, and you wanted to find the total weight of all rabbits over 2 years in age, you couldn't do it simply by looking at a list or lists coming out of the submodel.

The satellite submodel is a way of making this possible. It is a submodel with a single role arrow pointing at it (in contrast with an association submodel, which has two role arrows). At mostthere will be one instance of this submodel for every instance of the parent submodel (the one the role arrow comes from). But, as with a conditional or an association submodel,

you can place a condition model element (the questionmark symbol) inside it, and this can limit the number of instances to be some subset of the maximum number possible. The condition you insert is thens pecified to restrict the subset to that which you require.

Using a submodel to specify different time bases for different parts of a model.

By default, all parts of your model 'tick' at the same rate, as specified by the 'Update every...' value in the Run Control dialogue window. However, you will sometimes want to get parts of the model updated less (or more) frequently than others. For example, you may have a model containing both trees and a crop. The crop you want to grow on a weekly basis, so you can capture its response to rainfall patterns, pest outbreaks etc. The trees grow slowly, and there is no point at all in calculating tiny increments on a week-by-week basis. Conversely, you model may include a fire spread submodel, which is triggered only very occasionally but then needs to simulate the spread of fire at very short time steps.

In order to specify this, you need to ensure that the component of the model with a separate time base is in a separate submodel. You then specify the time base for this submodel (and, by default, any inside it) in the submodel properties box.

When you come to run themodel, Simile then realises that there is more than one time base for the model, and adds one or more extra 'Update every...' in the Run Control dialogue window. The user of the model then needs to specify each one separately, in terms of the model's unit of time. In the above example, the value 1 (year) and 0.02 (years, equals roughly 1 week) could be specified for the two 'Update every...' values.

Modelling approaches supported by Simile

Modellers frequently classify their model as being one type of another: 'differential equation model', 'matrix model', 'individual-based model', and so on. Simile is capable of handling most of these model types - sometimes directly, sometimes with some recasting. The aim of this section is to consider some of the more common model types, and how they are handled in Simile. It should be remembered, however, that a model in Simile can combine what would normally be considered distinct and non-combinable modelling approaches. Thus, in Simile, it is quite possible to have a model that combines, for example, differential-equation, matrix, and individual-based modelling.

System Dynamics

System Dynamics is a widely-used graphical notation for representing continuous systems. A System Dynamics diagram typically contains 4 main types of element: compartments, flows, variables and influences. Compartments represent storages of material, and the flows represent movement of the material into, out of and between compartments. Simile was designed as a 'System Dynamics plus objects' language, so naturally it is straightforward to represent standard SD models in Simile.

Some other visual SD modelling packages, such as Stella, introduce new symbols which extend the SD language (such as 'oven' and 'boxcar'). Simile does not have these, but their behaviour can be reproduced using Simile's core set of model-building elements.

Differential/difference equation modelling

System Dynamics is nothing more than a palatable front-end to a set of Differential-Algebraic Equations (DAEs): i.e. a set of differential equations, with a set of subsidiary algebraic equations for defining intermediate and rate quantities. Each compartment is a state variable, and each flow contributes to the rate-of-change expression for the associated state variable(s).

Therefore, implementing a published differential-equation model in Simile is straightforward: you simply add one compartment for every state variable, give each compartment a single inflow, and enter the differential equation as the flow expression, having first added the required influence arrows.

Since Simile uses simple Euler integration for numerically solving its set of differential equations, handling sets of difference equations is no different: the only difference is when you come to run the model, when you set the time step to 1 rather than some small value.

Disaggregation

'Disaggregation' refers to modelling in which some component is divided into a number of parts. For example, a lumped (non-disaggregated) model of soil water dynamics might use a single compartment to represent the amount of water in the soil. A disaggregated version of the same model might divide the soil into a number of layers, and represent the amount of water in each layer. Or: a population may be divided into age-classes; an area may be divided into subareas; a single 'vegetation' component may be divided into the separate species. It is a very common and important modelling technique, required to capture dynamics that would be lost in a lumped model of the same system.

In conventional programming and in the other visual modelling packages, disaggregation is handled by declaring certain variables to be arrays rather than scalars (single-valued variables). This is a tedious approach, reflecting the design approach of the earlier programming languages (Fortran, BASIC), which is difficult to read visually.

Simile encourages a quite different approach. You use a submodel to represent one of the disaggregated elements: a single soil layer, a single population class, a single subarea, or a single vegetation species. This submodel is then made into a multiple-instance submodel, so that it now represents all the soil layers, all the population classes, etc. Finally, you add the links that relate one element to the next: e.g. the flow of water from one layer to the next.

Matrix models

Models based on matrix algebra are frequently used in ecology for modelling population dynamics, with the population disaggregated into age- or size-classes. Simile does have the ability to do matrix calculations with arrays but, as mentioned above, a more appropriate way of handling such problems is to view a class as being an object, and then specify that there are as many instances as there are classes. Class attributes, such as age-specific fecundity and mortality rates, can then be expressed as attributes of each instance.

Individual/object-based modelling

Modelling the individuals that constitute a population is an extreme form of disaggregation. It is being increasingly recognised as a highly effective approach, for two main reasons. First, it enables the modeller to capture interactions critical to system behaviour that are lost in any more aggregated approach. Second, it frequently is much easier to construct individual-based models, since the behaviour of and interactions between individuals are frequently quite simple, but can lead to complex patterns of behaviour of the whole population (e.g. ant colonies).

If you have a fixed number of individuals, then you can use Simile's fixed-membership submodel. If, as is likely to be the case, the number of individuals in the population changes dynamically, then you would use Simile's population submodel, adding the three symbols needed to specify the initial number of individuals in the population, the way in which new

individuals are created (by migration in or by reproduction), and the way in which individuals are killed off.

Object-oriented modelling

The term 'object-oriented' has a formal meaning in software engineering: it is not just 'modelling with objects' in the sense of individual-based modelling. Rather, it reflects a commitment to a number of principles which together characterise the object-oriented approach, including message-passing, encapsulation (hiding internal detail), inheritance (from class to subclass), and polymorphism (the same procedure can operate on different data types). There is a strong movement towards the adoption of object-oriented software engineering approaches in ecological modelling.

It may come as some surprise, therefore, that not only does Simile not incorporate most of the characteristic features of the object-oriented approach, but that we have deliberately decided not to incorporate them. This is a controversial area, which we won't develop here. But briefly:

- the message-passing paradigm is inappropriate for systems based on differential/difference equations;
- encapsulation is just what modellers does not want: they should have access to any attributes of any object;
- inheritance, and a class hierarchy, can rapidly become extremely messy when the class is a designed model, and the subclass is some modification of this design;
- polymorphism has no particular role to play in modelling.

Rejecting encapsulation does not mean that we are against modularity or re-use of components: quite the opposite. We are just against the principle that you really should not know what is inside a modelled component.

Simile does, however, support certain object-oriented design concepts. A Simile diagram with submodels corresponds closely to a UML class diagram. The Simile notation of placing one submodel inside another to indicate containership corresponds closely to a composition association. And Simile provides explicit notation for an association between classes.

Spatial modelling

The term 'spatial modelling' refers to a particular form of disaggregation, in which an area is divided into a number (often a large number) of similar units: typically grid squares or polygons. The model may be linked to a GIS for data input and display. The transition from non-spatial to spatial modelling is often considered to be pretty significant, and there are a number of modelling packages that advertise their spatial modelling capabilities: indeed, many are labelled as landscape or landuse modelling tools.

Simile is rather odd in that it can do spatial modelling - in fact, it can do it rather well - while having no explicit spatial modelling constructs! All you need to use are the standard Simile model-design elements. It is only the use of maps for displaying model results (and potentially inputting information into the model) that reveal that it is a spatial model - and all the Simile input/output tools (helpers) are quite independent of the basic simulation engine.

In Simile, a spatial unit is just like any other unit. You define a single instance in a submodel, and then specify that you have many instances. The only difference now is that each instance will have two attributes specifying its x and y coordinates. These enable any display tool to locate each unit in the appropriate part of the display (map). They may also be used in spatial reasoning within the model: e.g. for working out the distance between two spatial units to calculate seed dispersal or shading.

The model itself does not 'know' if a spatial unit is a square, a polygon, a hexagon or whatever (in fact, it does not even know that it contains spatial units). It is up to the modeller to model each type in a way which is consistent with what is intended. Thus, if the units are grid squares, then they should all have the same value for an 'area' attribute, and the rule for defining which units are neighbours of which other units can be defined in terms of the column and row attributes. If they are polygons, then both the area of each unit and its neighbours will (typically) be defined in a data file. And in either case, the user of the model would need to choose a display tool that is appropriate to the type of unit being modelled.

Modular modelling

The term 'modular modelling' usually refers to the use of interchangeable components (or modules) in a model. The component may be a single equation, but typically it is a large component: for example, a plant submodel or a soil water submodel. There have been calls for the development of modular modelling approaches for some two decades, and some working systems, motivated by the advantages that this would confer on the modelling process in terms of model construction, testing and reuseability of components. In addition, a major motive for the adoption of object-oriented software engineering approaches has been its support for modularity in modelling.

The purest form is 'plug-and-play' modularity, in which the interfacing between a module and the main model is pre-defined (like the pins on an integrated circuit chip). All the modeller needs to do is to load the module, and it is automatically part of the model. Simile enables you to o this, as a two-step operation. First, you load the module (a Simile model, loaded into a submodel window). Then you select an Interface Specification File which defines the links between the submodel and the rest of the model. This approach has considerable merits: it means that the same Simile model can be inserted as a module in a range of other models, with different interfacing for each one.

At the other extreme, Simile supports 'free-form' modularity, in which it is entirely up to the modeller to decide how the inserted submodel links to the rest of the model. This means that the modeller has access to a much greater range of models to use as submodels - ones that were developed with no intention that they be used as a submodel in someone else's model. This removes the need for careful defining of interfaces which plagues current modular (and indeed object-oriented) systems.

Population dynamics

You will probably by now appreciate that you have a variety of options when it comes to modelling population dynamics. You can adopt a lumped approach in System Dynamics, using a compartment to represent population size, and flows to represent demographic processes of reproduction, migration and mortality. Or, you can adopt a disaggregated System Dynamics approach, using a compartment-flow structure to represent the dynamics of one age/size/sex class, embedded in a multiple-instance submodel to represent all the classes. In this case, you will need to have additional flows for modelling the movement of individuals between age or size classes. Or you use a population submodel to represent every individual in the population, adding in rules for specifying the creation of new individuals and the death of existing ones. In this case, it is optional whether your model contains any System Dynamics at all: you might decide to use a compartment-flow structure to represent (e.g.) the body weight of each individual, but you don't have to.

Input/output (display) tools

Any simulation model needs some method for reporting on the values of modelled variables. When the model is programmed in a conventional programming language, the programmers

normally write program utilities that draw graphs, produce tables etc, or export these values to a file from whence they can be picked up in a spreadsheet or graphics package. Generic modelling environments usually provide a set of displays, built into the package: the modeller just has to accept the ones provided.

Simile is unusual, if not unique, in that all the input/output tools are interpreted programs totally independent from the core Simile package. Simile recognises that they are present during a particular session purely because they reside in a certain directory. Any Simile user can modify the tools provided, or develop their own, totally independently of the Simile developers.

Such tools can be customised for the particular models that the user constructs. For example, if you are modelling elephants, you could develop a tool that shows elephants wandering across an area. It may be that you are the only person who ever wants to use this tool. It may be that you know of other elephant modellers who would appreciate such a display - in that case, you just send them a copy.

Each display tool is programmed in Simile's native user interface language, tcl/tk. This is an interpreted language (you can look at the programming of the existing tools in the Simile installation), and has a C-like syntax. You certainly need to have, or have access to, some programming expertise to modify existing ones or develop your own, but that effort is required for just one small part of the whole modelling activity, not for the whole model. Moreover, any investment you make in this can benefit not just yourself, since you can share it with the rest of the Simile community.

Working in the Simile environment

Constructing the model diagram

Simile provides a toolbar containing:

- the symbols you can use in constructing the model (corresponding to the model elements described above);
- tools for editing the model diagram (like deleting elements); and
- normal application tools, such as file open, save, etc.

To construct a model diagram, you first click on the tool for the required symbol, e.g. a compartment. This puts you into that mode, and you remain in that model until you select another tool. You then click anywhere in the model diagram window to place a symbol or, in the case of submodels and arrows, you drag the mouse from a start position to an end position. Every diagrammatic element is coloured red until you have entered a value of equation for it: this indicates that the model is unrunnable because some required mathematical information is missing.

Entering values and equations

At any stage during the process of making the model diagram, you can enter a value or an equation for a model element. You do this by going into Pointer mode, then double-clicking on the appropriate element. This opens up an Equation dialogue window into which you can type the value or equation. The entry is checked for syntax and for use of the inputs specified for that model element by influence arrows on the diagram. Once you have provided a value or equation for a model element, it turns from red to black on the model diagram. When all elements in the diagram are black, the model is runnable.

Calling up display tools

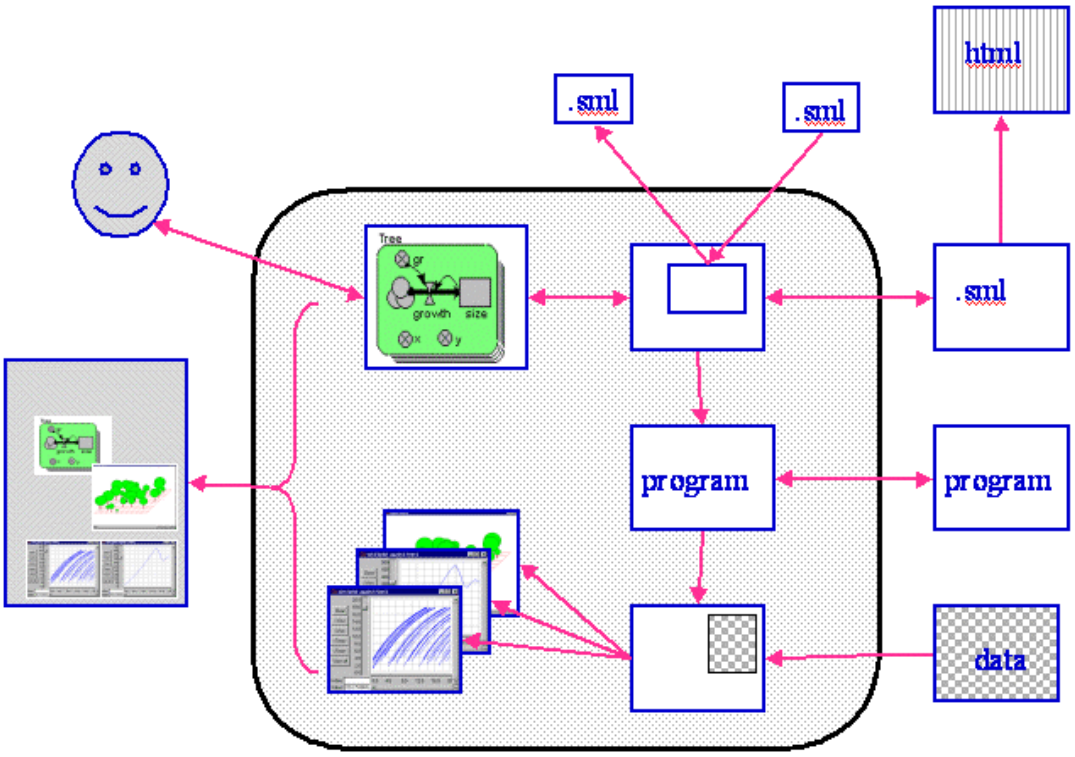
In order to see results from the running model, you need to call up one or more display tools (also known as 'helpers'). There are numerous tools available, such as graphs, tabular output, maps, and lollipop diagrams (designed for showing trees in a forest stand). Each tool requires you to specify the variable or variables needed to configure it: you do this by clicking on the variables in your model diagram when instructed to do so. You can call up displays for any variable, including flows and intermediate variables.

Running the model

Currently, Simile provides a simple run-control interface, allowing the user to set the time step for numerical integration, the display interval, and the run duration. Runs can be continued, or reset to begin a new run.

Simile architecture

The following diagram shows the main components of Simile, and how they relate to each other. This is something of a simplification of the real story, but is sufficient for a first encounter. The large round-cornered box contains items that we can consider as being part of the Simile environment. These interact with items, mostly files, that can be handled independently of Simile.



We begin at the middle-left. The user interacts with Simile through a graphical user interface (GUI) to edit the model diagram and equations. In the process, Simile builds up an internal representation of the model, which may contain submodels. The model or a submodel can be saved to file (.sml extension) (arrows going to the top and to the right). Also, a model or a submodel can be loaded from file (arrows coming in from top and right).

To run a model, the internal representation is used to generate a program in tcl or C. (In fact, not shown here is that in C Simile first generates the source code, then compiles it as a DLL).

This program can also be saved to file: it is possible to run a previously-built model as a stand-alone executable without having access to the Simile environment. The program may then need to be combined with external data sets, containing parameter and tabulated values (if the user has specified that certain variables have their data supplied from a file). When the model is actually being run, Simile calls on various display tools to show the results of the simulation. The displayed results of the simulation may then be exported, along with the model diagram, to produce a postscript file that can be used to produce a handout or poster about the modelling exercise.

At the top-right of the diagram, you will notice that the saved model can be used to generate an html description of the model. This is handled by a program (written in Prolog) that is totally independent of Simile itself. It illustrates an important point of principle underlying the declarative approach to modelling: that it is possible to have a suite of independent tools that can process models, provided that there is an open format for representing the models in external files. The html generator is the first of many such tools, and ultimately we envisage that many groups around the world will be writing their own for processing Simile (or Simile-like) models in new and useful ways.

Example models

Probably the best way of getting to know Simile is to see some examples. The rest of this section is a shop window to illustrate the range of subject areas that Simile can be applied to, and how various types of modelling problem are tackled in Simile.

Some sections deal with a single model; others take you through a range of alternative models so you can compare different solutions to the same problem. Some of the models are very simple, skeleton models, designed to illustrate a concept or technique; others are serious research models included here to show you what it is like to use Simile for real modelling.

In general, most of the examples are presented at the model-diagram level, sometimes with sample output from the running model where that helps your understanding of what the model is doing. I have usually avoided including equations, unless they are very simple or unless you would miss some important point. The point of this section is to give you a feel for Simile and how it approaches things, not to go into full details for each of the models. We have plans to set up a web-based catalogue of Simile models - including the ones presented here - and that will include full mathematical description of each model plus the model itself ready for loading into Simile.

I expect that most people will probably want to dip into this section, looking at models that relate to their area of interest or that illustrate a technique you want to know more about. That's fine: the following table should help you find what you want. There is, however, a general trend from simple to more advanced Simile concepts and techniques, so you may need to refer back to earlier examples if I seem to be taking something for granted.

Table of example models

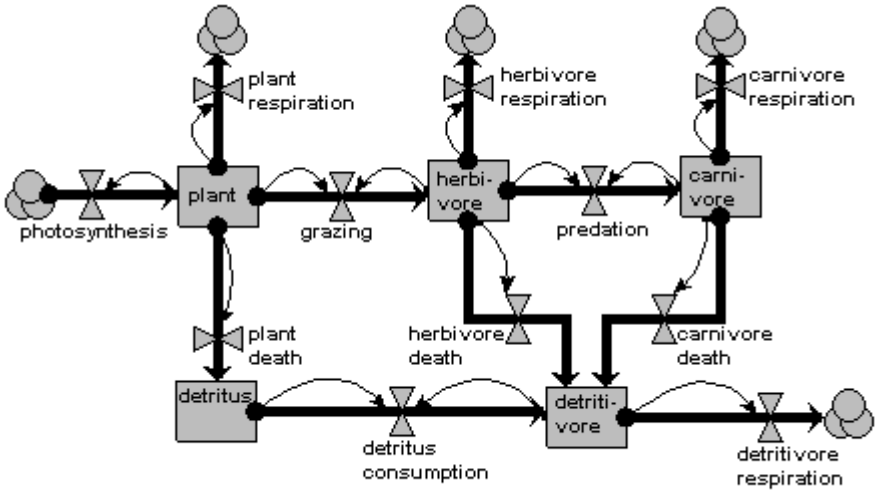
Topic	Simile techniques	Comments
Ecosystem energy flow	System Dynamics	Illustrates basic System Dynamics diagramming in Simile
Population dynamics	Fixed-membership, association and population submodels; arrays	Comparison of lumped, age-class and individual-based models; and of alternative ways of implementing age-class models in Simile.
Modular modelling (ecosystem primary production)	Fixed-membership submodel; modular modelling	The context is a published model of ecosystem primary production, including a soil water submodel. It's used to illustrate 'plug-and-play' modularity.
Landuse change	Fixed-membership submodel; condition submodel; association submodel	A simple skeleton model of landuse change, illustrating some more advanced Simile techniques. Could be the basis for a serious research model.
Modelling people and landuse: the FLORES project	Fixed-membership, conditional, association and satellite submodels. Nesting of submodels.	A complex model, intended for investigating policy options. Includes the full range of Simile techniques, but only described in general terms here.

Ecosystem energy flow

This model represents the flow of energy through the different components of an ecosystem. It is a classic compartmental model, conforming to conventional System Dynamics notation, and could be represented in any of the standard System Dynamics visual modelling packages.

The model has 5 compartments, representing the energy content (joules) in each of 5 ecosystem compartments: the primary food chain consisting of plants, herbivores and carnivores, and a detritus chain. The flows show the processes that move energy from one compartment to the next. The overlapping circles represent clouds, signifying a System Dynamics 'source/sink': in effect, a compartment outside the system, whose value we are not interested in. The thin arrows represent influences, and indicate which compartments are deemed to influence each flow: i.e the flow rate is dependent on the value in the compartment.

Note that, though very simple, there are still significant design decisions indicated here. For example, the feeding flows are shown to be dependent on the amounts in both the food compartment and in the feeding compartment. Also, we assume here that dead plant material goes into a detritus compartment, because of its relatively long residence time, whereas dead animal material is consumed immediately by the detritivores.

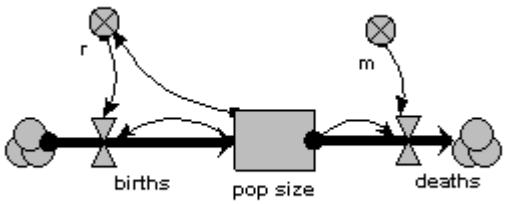


Population dynamics

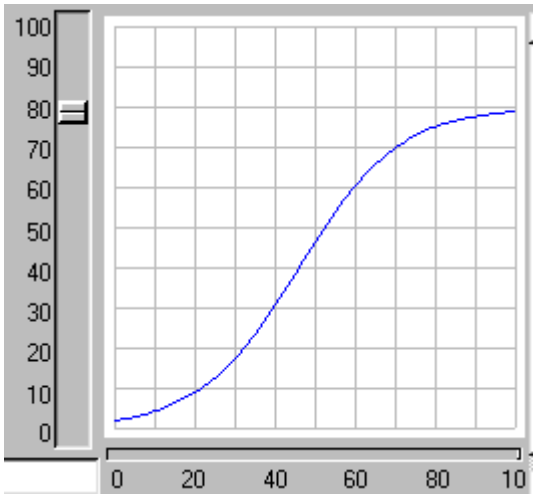
The following sequence of 5 models illustrates two things. First, it shows that one problem - that of modelling changing population size - can be approached in a variety of ways, based on quite different methods for representing the population. Second, it introduces you to a range of the constructs that Simile provides for modelling, including the 'population submodel', array variables, and the 'association submodel'.

Population dynamics example 1: A lumped population

In this model, the population is represented by a single compartment, representing the total number of animals in the population or the population density (number per unit area). Reproduction is assumed to be density dependent (the reproductive rate pre individual, r, declines as population size increases), whereas mortality is not: the mortality rate per individual, m, is independent of population size.

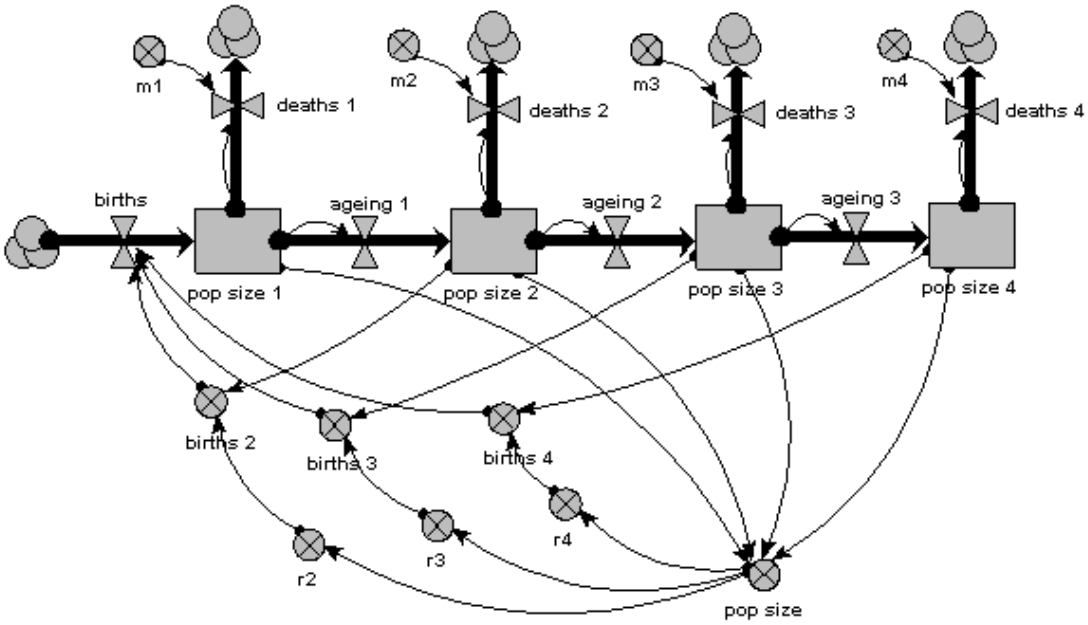


When the model is run, it shows the expected logistic form of population growth. When there are 80 individuals in the population, the birth rate balances the death rate.

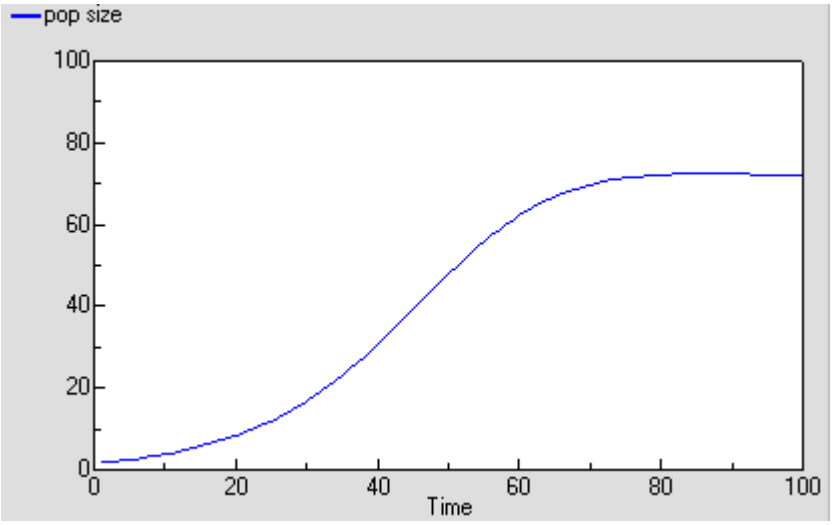


Population dynamics example 2: Age classes, with a separate compartment for each age class

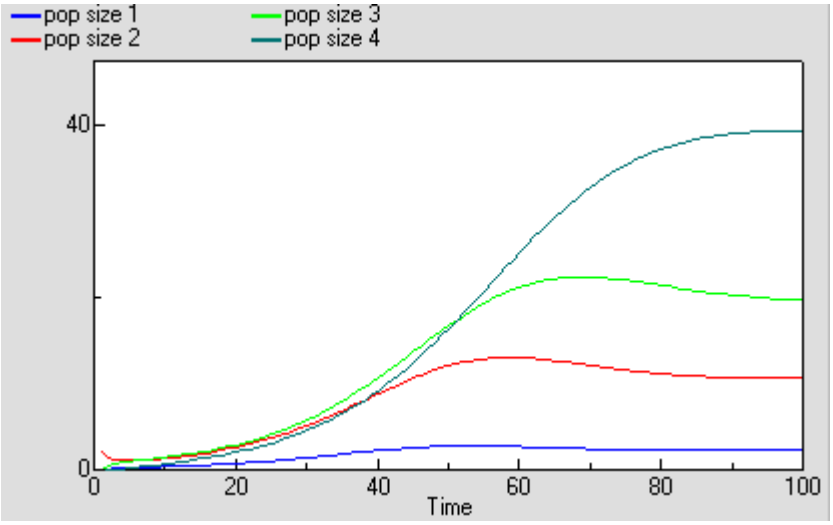
The same population is now represented by 4 compartments, one for each of 4 age classes. The first one represents the first year group, the second represents the number of animals aged 1-5 inclusive, the next animals 6-15 inclusive, and the fourth for all remaining animals. As before, the mortality rate per animal is independent of population size, but now depends on its age (there is a different m value for each age class). The reproductive rate per individual is still influenced by total population size, as well as now being different for each age class. Note that the first age class does not reproduce, so we simply do not include a births term for it.



As before, we can now run the model and plot the dynamics of the total population size: this is in the variable called 'pop size', which is simply the sum of the population sizes for each age class. This again shows logistic growth: it will differ in detail, since the single-valued parameters we had before are now different for each age class.



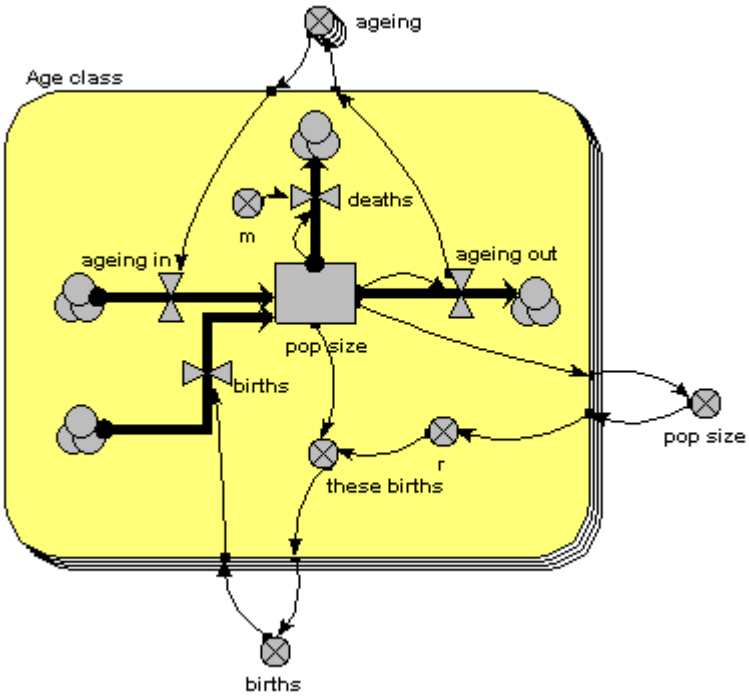
But we are now able to inspect the dynamics for each of the 4 age classes. We see that the stable total population size is reflected in stable numbers in each of the 4 age classes, with different numbers in each class reflecting their different population parameters. Notice that there is some overshoot before the stable values are reached.



Population dynamics example 3: Age classes, using a multiple-instance submodel
The above two examples are expressed in conventional System Dynamics terms. They set the scene for the next 3, which are now expressed in terms specific to Simile.

The first age-class model had just 4 age-classes, but already we can start to see some problems. First, the diagram is starting to get a bit messy: this would get worse if, for example, the mortality rates were also density-dependent. Second, any change to one of the basic equations, such as that for working out the birth rate per individual, has to be done separately for each age class. We can easily see that this approach does not really scale up to a larger number of classes: for example, if we decided to have one-year classes for the whole population. What we would like to have is a way of specifying once how an age-class works, then get this repeated automatically across all age classes.

In other visual modelling environments, such as Stella and ModelMaker, this is done by allowing the user to specify that some variables are in fact arrays. In Simile, the approach is to specify the compartment-flow structure in a submodel for one age-class in terms of scalar (single-valued) variables, then specify that the submodel has multiple instances, one for each age class.



The main thing to note about this diagram is that it now has just one compartment. This is in a multiple-instance submodel, called 'Age class', with 4 instances, one for each class. The variable 'pop size' inside the age class submodel represents the population size for each age class; the variable with the name 'pop size' outside the submodel represents the total population size (and is the sum of the population sizes for each age class).

What previously was a single ageing flow connecting two successive compartments is now two flows: one for the number leaving a compartment through ageing ('ageing out'), the other for the number entering a compartment from the class below ('ageing in').

How does the 'ageing in' flow for one age class know what the 'ageing out' flow is for the class below it? This is clearly central to the whole model: everything else can be done on a class-by-class basis, but this involves some form of connection between classes. And we can't do that inside the submodel, since each instance can only know about its own values, not those of other instances. The answer is to take the 'ageing out' values outside the submodel, and put them into an array ('ageing'). This array is them made available to the 'ageing in' flow, which extracts the value for the previous class.

Skip over this on first reading!

The actual formula for 'ageing in' is (almost):

element([ageing],index(1)-1)

which makes use of two built-in functions. **index(1)** returns the instance number for the current instance of the submodel, so '**index(1)-1**' refers to the previous instance. **element([A],I)** selects element I from the array I. The only extra thing we need is a conditional statement to avoid doing this for the first instance (since there is none before it), so the above expression is made conditional on the instance number:

if index(1)>1 then element([ageing],index(1)-1) else 0.

There is a births inflow for each age class. This is required since we are modelling a generic age class, but in fact the equation for this flow sets it to zero for all classes except the first. This flow is derived from the total births, which is held in the variable outside the age class submodel: that in turn is calculated from the sum of 'these births', which is the number of

births for each of the age classes. Remember in the previous model how the first age class did not reproduce, so we simply left off the variables for it? We can't do that now: instead, we set the parameter value r to zero for the first class.

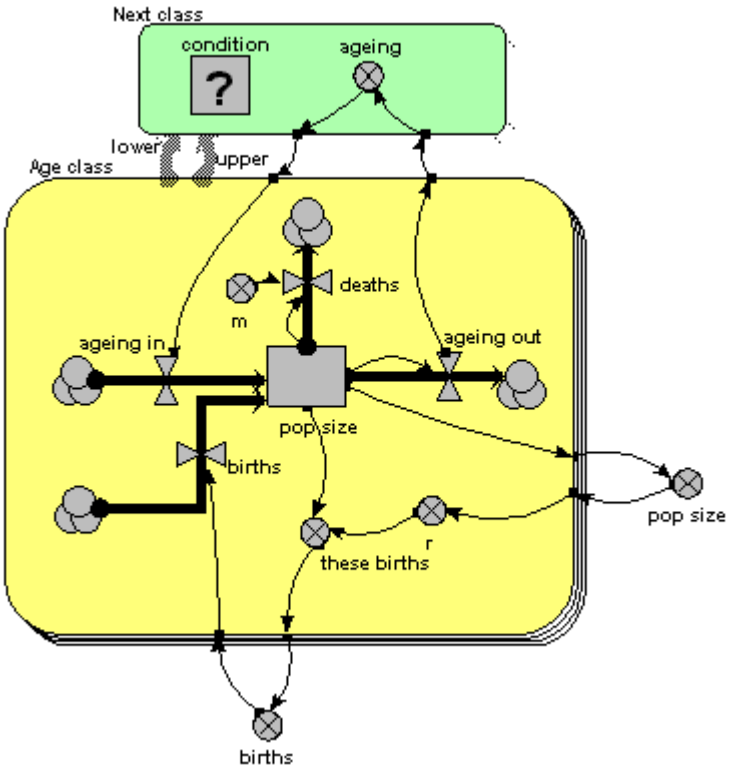
The behaviour of this model is identical to the previous version, in which we had a separate compartment for each age class. We have not changed it mathematically: merely the way it has been implemented.

Population dynamics example 4: Age classes, using a multiple-instance submodel and an association model

The previous method is a big improvement: it is a much more concise way of representing multiple age classes, and exactly the same model can be used regardless of the number of classes we have: in other words, it scales up to larger problems. However, it suffers from a couple of disadvantages:

- First, there is a problem with the amount of data flowing around. Let's say that we have 100 classes. Then Simile has to handle 100x100 values, since it has to make the array holding the 100 values available to each of the 100 classes. This is actually manageable, but still involves a lot of unnecessary computation (when we consider that in fact that there are only 100-1 = 99 interactions between classes, one for each pair of ageing transitions).
- Second, our model diagram does not really convey the fact that there is a special relationship between classes, namely that one class is immediately above the previous one. Class 2 is above class1, class 3 is above class 2, and so on. There is no connection between class 1 and class 3. Since this is an important feature of our model, and since one aim of the model diagram is to convey information about the model design down to a certain level, it would be nice if the model diagram showed that some form of relationship existed between classes.

Simile's association submodel provides a way for tackling both problems. This is quite a big step in your understanding of Simile if you have not come across it before, particularly if you have no background in relational databases or the concept of 'association' in object-oriented software engineering. So all I will do here is to present the model diagram and talk you through that: an explanation of the equations used can wait until you actually come to use it.



The only difference from the previous model is that the single array variable, holding all the ageing flows from one age class to the next, has been replaced by am ageing variable inside another little submodel, called 'Next class'. This is made with Simile's submodel tool, just like any other submodel. It is, however, different in two respects from other submodels. First, it has two broad grey arrows pointing to it from the 'Age class' submodel. These indicate that there is an 'next class' relationship (or 'association') between one class (the lower one) and another (the upper one). Second, we see that the submodel contains a condition (questionmark) element: this is used to specify the condition under which the 'next class' relationship holds true. In fact, it holds true if, and only if, the instance number of one class is just one higher than that of another class: that condition is entered in the Equation box for the questionmark symbol.

What happens is that all the values for 'ageing out' is passed to the submodel, but only one value is passed back into the submodel for each class: the one for the class above it.

You will have noticed that the model diagram does not itself tell us just what relationship exists between classes. However, it does tell us that some sort of relationship exists between them, so the diagram has helped in communicating the nature of the model down to a certain level. This is rather like when we have a variable with influences on it: we can see that the influences exist, without seeing the actual equation used to capture their effect.

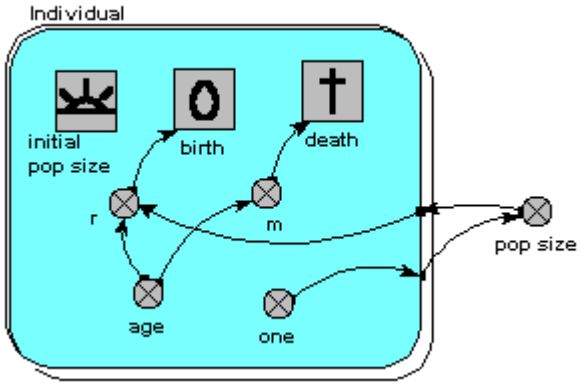
Population dynamics example 5: Using a population submodel

The above examples all share one thing in common: they represent a population in terms of the values for one or more state variables, there being as many state variables as there are classes in the population. Each state variable may be engineered to have integer values (on the basis that populations contain a discrete number of individuals), or to have real (floating point) values, if you are content to think in terms of population density (i.e. number per unit area) rather than absolute numbers. The number of state variables (hence the computational requirement) is fixed, and totally independent of the number of individuals in the population.

A totally different approach is to represent each individual in the population separately. This can be justified on the basis of what is considered necessary to adequately capture the factors driving population dynamics. For example, if we know that social interactions between individuals is important, then we need some way of capturing these: it may be difficult to approximate to their effects in a lumped or class-disaggregated model.

Since we are interested in population dynamics, building an individual-base model will require that we create new individuals as time goes on, and remove existing ones. Thus, the amount of memory required to manage this population will change as the simulation proceeds. If the model is implemented by programming, the difficulty of doing this increases substantially as we move from a fixed number of state variables to a dynamically-changing list.

In Simile, it is very easy to create an individual-based model. You simply use a Simile submodel to define how one individual works, then declare this submodel to be a population submodel. You can then add model elements to hold the equations or rules for the creation of new individuals and the killing off of existing ones.



Here's a model which is the individual-based equivalent of the age-class model with density-dependent reproduction. We make a submodel and call it 'Individual' (following the principle that the label for a multiple-instance submodel should reflect the single object it references). It is specified as a population submodel by simply choosing this option in the submodel properties box. A population submodel is drawn with shadow lines on the top-left and bottom-right, to distinguish it from the fixed-membership multiple instance submodel we have seen so far.

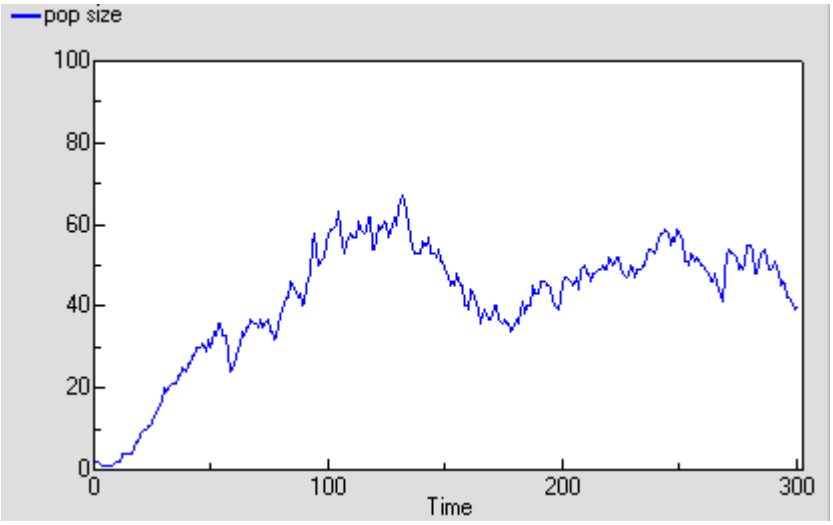
It has symbols for representing the initial number (i.e the number created when the model is initialised), an expression for creating new individuals by reproduction of existing ones ('birth'), and an expression for deciding when an individual dies ('death'). For consistency with the notation of the earlier models, we have two parameters, r and m, with similar meaning as before: r represents the probability that an individual will reproduce in one unit of time, and m represents the probability that an individual will die. Both r and m depend on the individual's age (calculated simply as the difference between the current simulation time and the time when the individual was created). r also depends on the size of the population (negative density dependence). The population size is calculated as the sum of the value 1 (held in the variable 'one') over all the individuals. Note that population size has to be outside the individual submodel, since it is an attribute of the whole population, not of any one individual.

This model is now a stochastic one: the parameters r and m are treated as probabilities rather than deterministically generating changes to the population. We could have handled these deterministically - for example, creating one new individual when we had accumulated

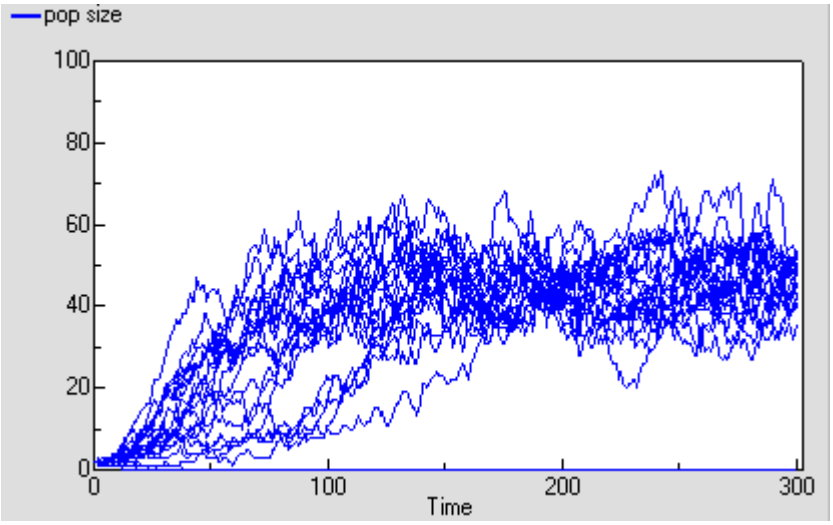
sufficient 'credit' - but a stochastic interpretation is more intuitive. We could also have made the previous models stochastic (by sampling from a distribution to get the annual reproductive inputs and mortality losses), but then we would have lost the link with standard 'textbook' models of population dynamics.

You may be surprised to see that there are no compartments here. We could have them: for example, we may want to model the change in body weight of each individual (or, say, height of each tree), in which case we could have a compartment and associated flows inside the submodel. We could equally well choose to have a complex System Dynamics model if, for example, we have a sophisticated, physiologically-based model of tree growth and we want to model a stand of trees on that basis. But this example shows that we do not have to include compartments and flows in a Simile model. Nevertheless, we can still use part of System Dynamics notation: the variable and the influence arrow.

The following graph shows one run with the model. It's run for 300 time units, since the reaching of an asymptote after 100 time units is less clearcut with this stochastic model. Even then, its stochastic nature means that there is considerable variation even once it has reached balance.



The following diagram shows the result for 25 replicate runs. The basic logistic growth can be discerned. It is also clear that the stochastic nature is more than simply variation around this line: some runs take quite a while to 'take off', and in fact in some cases the population goes extinct.



Conclusions on population modelling in Simile.
The above 5 examples have demonstrated a number of important points.

First, the population modeller can construct very different models based on more-or-less the same biological assumptions. The above models all have negative density-dependence for reproduction and density-independent mortality, but differ in their degree of disaggregation and the age-dependency of population processes. The modeller should be aware of these choices, and ideally be able to justify a choice vis-a-vis the alternatives.

Second, mathematically-identical models (the 3 age-class models) can be implemented in different ways. These do not affect the results produced, but do affect the ease of working with the model and the ease of communicating the model to others. Again, the modeller should be aware of such choices.

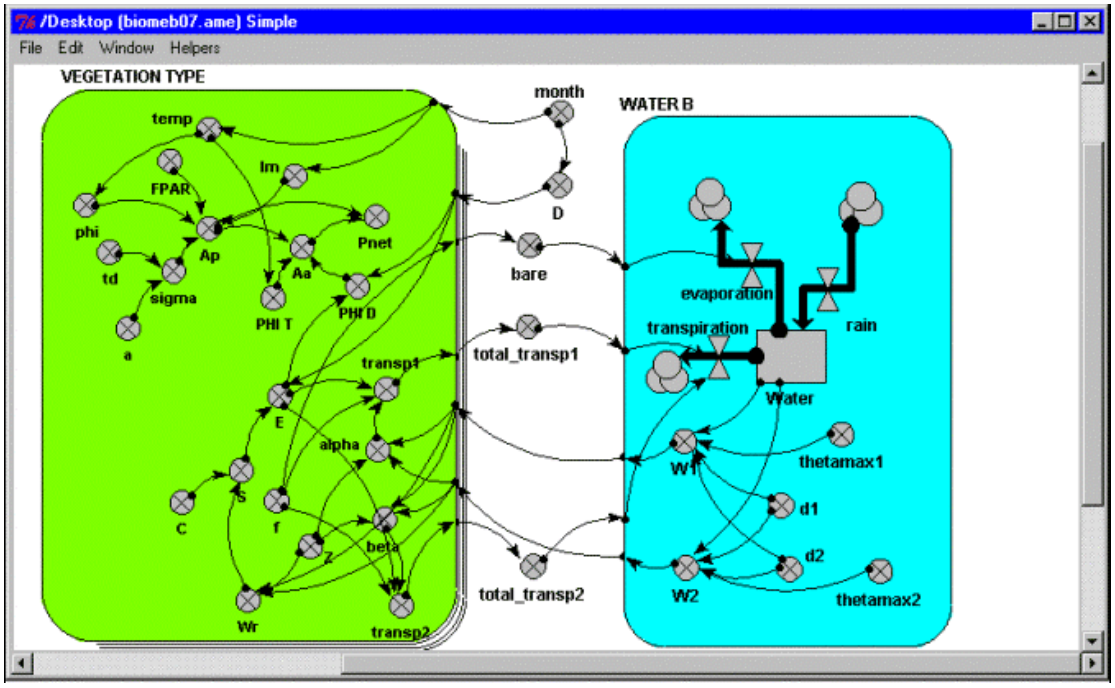
Third, Simile is unusual - if not unique - in enabling these very different models to be constructed within a single, easy-to-use environment. This minimises the amount of effort the modellers have to put in to explore these alternative approaches, compared with having to use different packages, or implementing the different models in a programming language.

Finally, we can broaden our vision beyond the modelling of one population. Usually, a population model will be a component of some larger system: the food on which the population depends; other predator populations; various abiotic factors; possibly even human factors (for example, in farming systems or wildlife conservation). Any of the population models we have considered above could be integrated into a larger, ecosystem model. Moreover, it could be wrapped up in an outer submodel envelope, and the alternative versions could be swapped in and out of the ecosystem model on a plug-and-play basis: another strong argument for enabling different modelling approaches to be implemented within a single modelling environment.

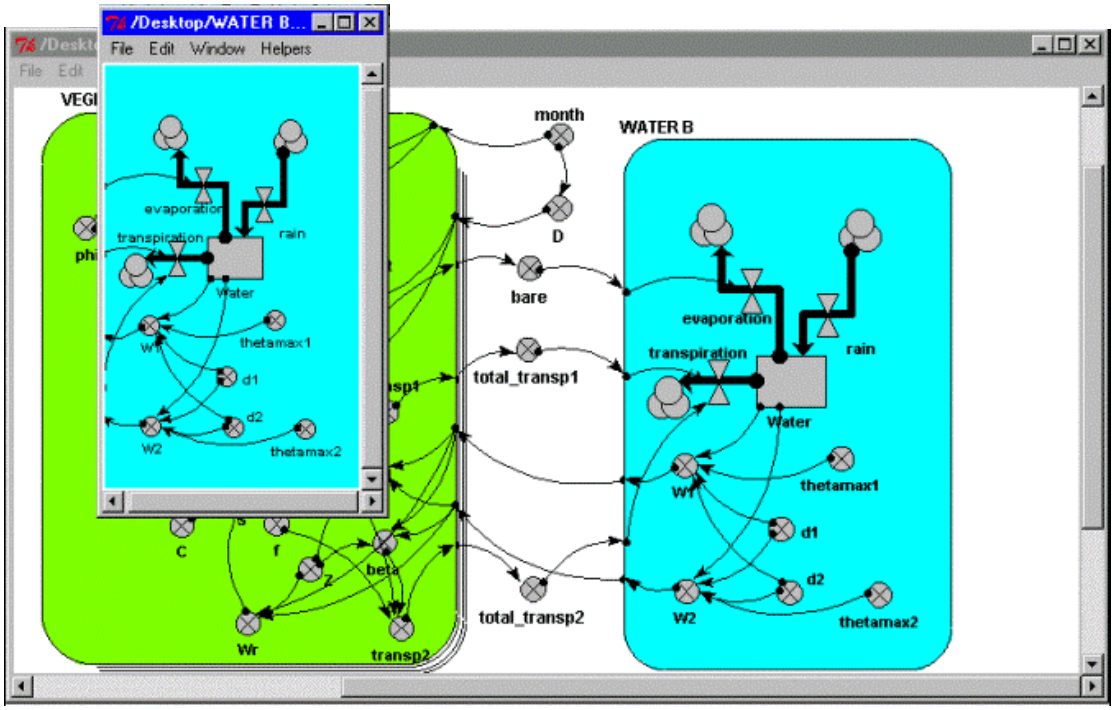
Modular modelling

The following example illustrates how Simile supports 'plug-and-play' modularity: i.e. the ability to replace one submodel with another using the same interface. The example is based closely on the 'Biome' model of Colin Prentice: it aims to calculate the net primary production for an area in Australia, for each of several vegetation types. The details of the model are not important here: what you should note is that the example is based on a real, published model, and that the model has two submodels. One of these - the soil water submodel - will be replace with an alternative in this exercise.

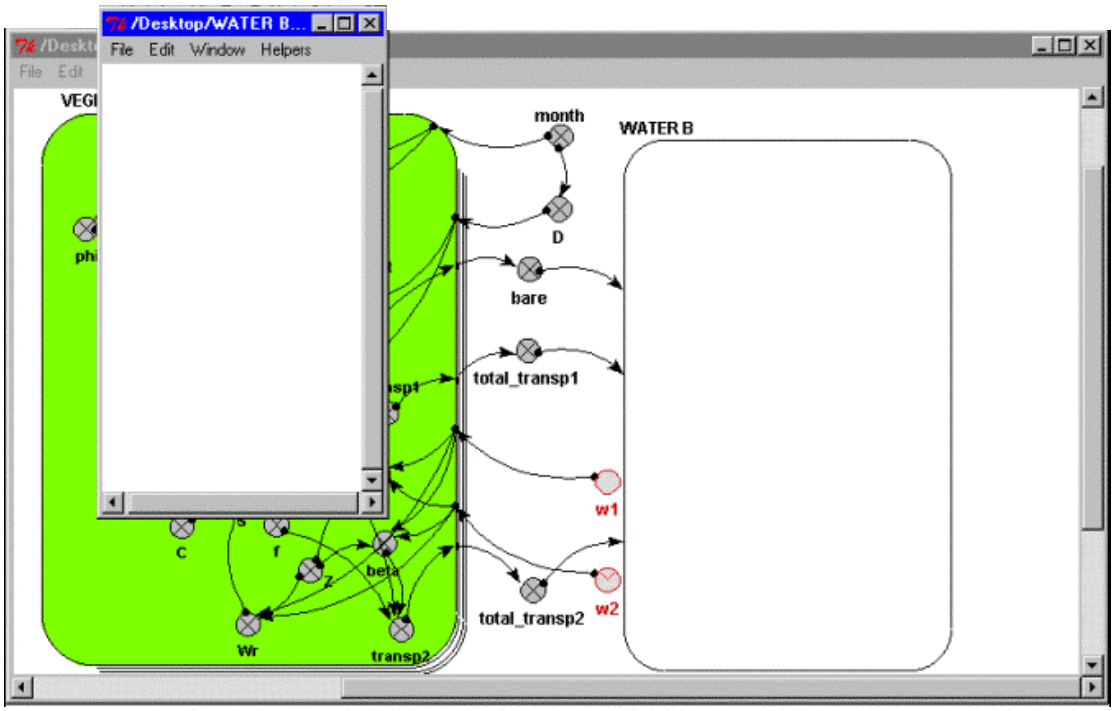
We begin with the diagram for the Simile implementation of the model. Note that, in the soil water submodel, there is just one soil water compartment, but that the amount of water is calculated for each of two layers. These two water content values (W1 and W2) are exported to the vegetation submodel. Conversely, the vegetation submodel exports values for the amount of bare soil, and the transpiration from each of the two soil layers to the soil water submodel.



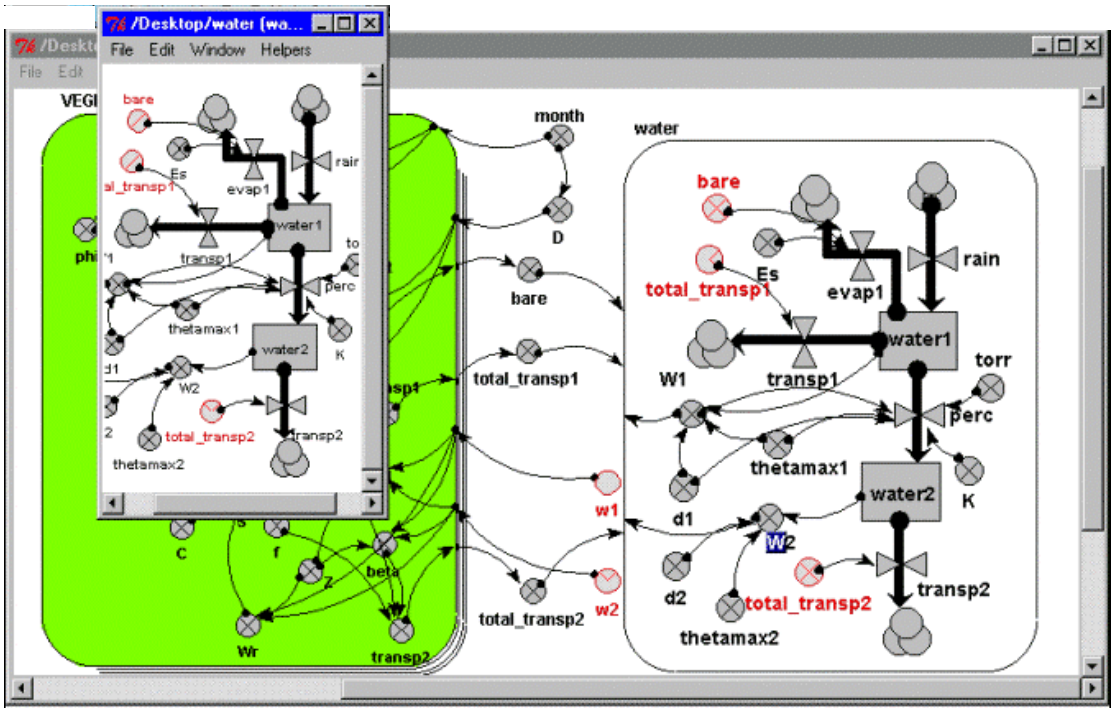
We now open up a separate window for the soil water submodel (by double-clicking on its boundary). We need to do this, since we need to access the File>New and File>Open commands in its menubar.



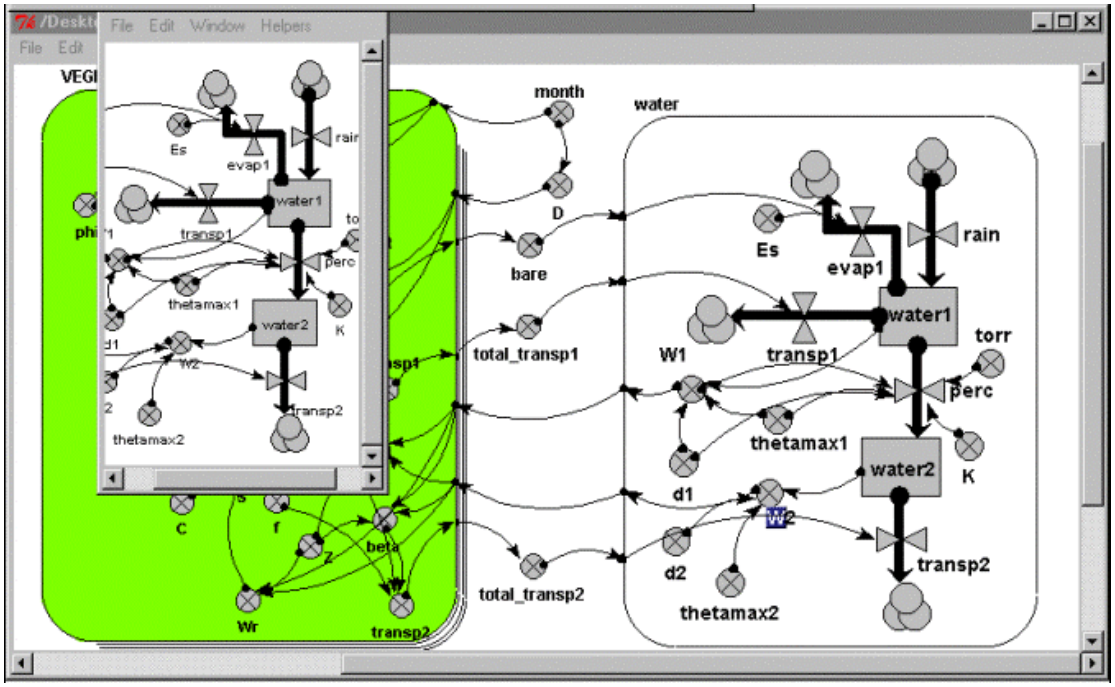
We now select File>New from the menubar for this submodel window. This removes all the model elements for the water submodel, leaving both the submodel window and the submodel in the main window empty. Note that Simile has automatically created two variables in the main widow for the two variables, W1 and W2, which are no longer available.



We now select File>Open in the submodel window, and load another soil water submodel. Note that this has a different internal structure (it has two soil water compartments rather than one), but has the same variables for interfacing with the rest of the model (W1 and W2; bare, total_transp1 and total_transp2). However, at this stage Simile has not made the links: the variables with these names are not linked up with influence arrows with the variables of the same name in the main model.



We now select Edit>Set interface spec from the menubar of the submodel window, and choose a file that defines the particular interfacing we require. On doing this, Simile automatically makes all the 5 links, removing the now-redundant variables w1 and w2 in the main model, and bare, total_transp1 and total_transp2 in the submodel.



We now have the new submodel inserted in the place of the original one.

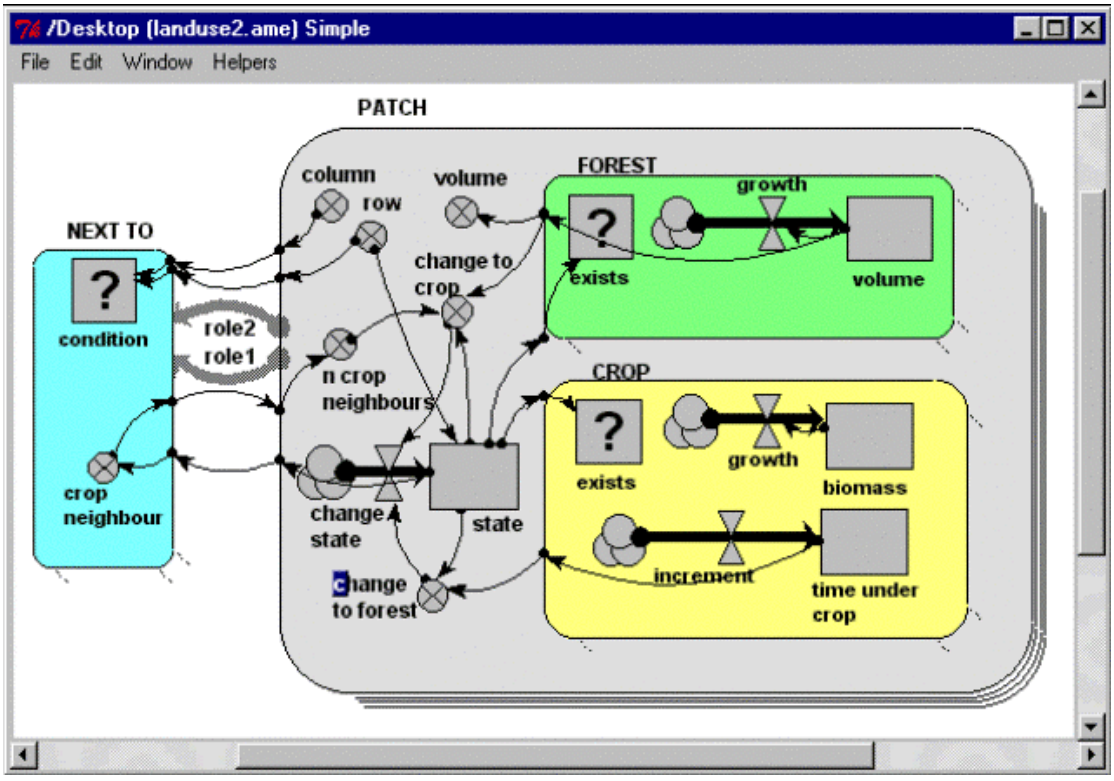
There are a couple of points to discuss. First, you might be wondering why this is a two-step process: you might think that 'plug-and-play' means that it should involve just one step (loading the new submodel). Well, we could have designed Simile to do that, but that would mean that every saved submodel would have to have stored with it information on its

interface to the main model. This is undesirable for two reasons. One is that, when you save a model, it should be possible to treat it as a stand-alone model in its own right. That argues against storing interface information along with the model. The second is that there is no reason why one submodel should not plug into a range of other models, with each potentially have a different interface. This could not be done if the interface was stored with the model. Therefore, we decided to separate out the interface specification as a separate form of information, saved in a separate file.

Second, this demonstration of plug-and-play modularity does not mean that that is the only method for modular modelling. It is quite possible to insert a new submodel in place of an existing one that has none of the required interfacing in place. In that case, the modeller can simply make the links between the inserted submodel and the rest of the model using the normal mechanism that Simile provides: constructing new influence links, and possibly adding in new intermediate variables. In fact, Simile is tolerant enough to support a mixed approach: it will automatically make whatever links it can using the interface spec file, and then allow the modeller to make any additional links by hand.

Modelling landuse change

This is an example of a spatial model implemented in Simile. It shows that Simile is capable of handling quite complex spatial behaviour - in this case, landuse change in which the neighbours of a patch influence the likelihood of transition from one landuse state to another - even though Simile has no built-in constructs for spatial modelling.



The model illustrates 3 types of submodel:

The PATCH submodel is a **fixed-membership, multiple-instance submodel**, and is used to represent the fact that there are many landuse patches. Each patch has row and column attributes, and each one is engineered to have a unique combination of values between 1 and 20, thereby defining each one's position on a grid. (But to drive home the point: Simile understands nothing from the labels 'row' and 'column': it is up to us, the modellers, to ensure that these are given values which can be used as grid co-ordinates.)

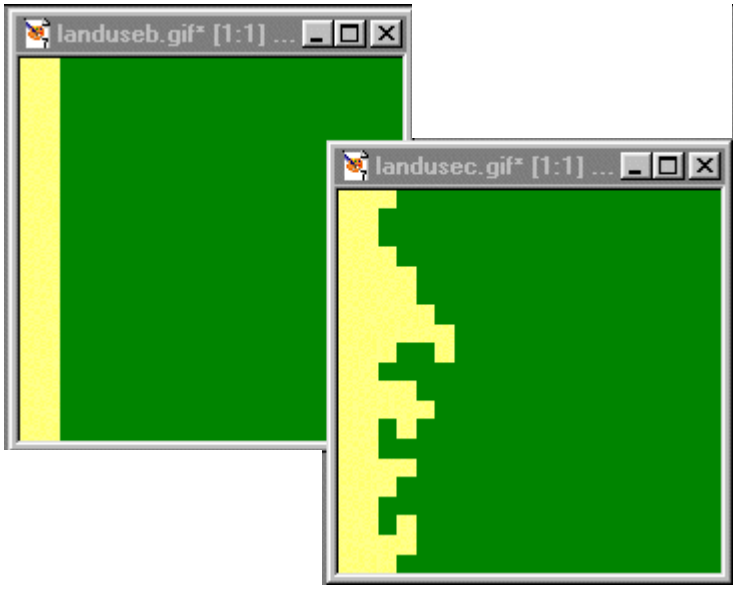
The FOREST and CROP submodels are contained inside the PATCH submodel. Therefore, each patch can (potentially) contain both a forest and a crop submodel. However, note that each one is a **conditional submodel**: see the question-mark 'condition' symbol in the top-left of each one, and note the little rows of dots leading from the bottom-right edges. This indicates that the submodel may exist in only some of the patches, not all of them. In fact, the conditions are engineered so that each patch contains only forest or crop, not both (but it would be quite possible for us to have some 'agroforestry' patches containing both, if that's what we wanted.)

Finally, the NEXT TO submodel is an **association submodel**, defining an association (relationship) between some patches and others. We can see that this is an association between patches by the presence of the two broad grey arrows (role1 and role2), pointing from the PATCH submodel to the NEXT TO submodel. As the name suggests, this association defines which patches are next to which other patches. Again, it has a condition

symbol, which is used to indicate under which conditions the association holds. We note that this has influences coming from the row and column variables in the PATCH submodel: the condition is true when the row and/or column value for one patch is one away from the row and/or column value of another patch.

The model works as follows. Each patch contains a state variable (the compartment 'state') which defines the state it starts off in. This is 1 for forest and 2 for crop. If a crop patch has been a crop patch for a certain length of time, then it is abandoned and reverts to forest (as mediated by the variable 'change to forest'). If the volume of the trees exceeds a certain amount and the patch has crop neighbours (this is why we need to know which patches are next to which others), then the forest is cleared and it changes to crop as the landuse (as mediated by the variable 'change to crop').

The following two figures show how the model behaves, using Simile's grid map display to show the patches on a spatial basis. The light (yellow) squares represent the patches under crop; the dark (green) squares represent forest. When users request this display, they are required to specify a variable indicating the column number for each patch, and the actual variable to be displayed: Simile then has sufficient information to lay the patches out in the correct grid-based manner. This display shows how Simile can handle spatially-referenced information, even though it has no built-in concept of spatial modelling.



Initially, most of the area is forest, with a band of cropping land on the left. After 40 years, patches of forest on the forest margin have been cleared, leading to a ragged edge to the forest boundary. In this particular run, the model was set up so that there was no reversion of cropped land back to forest, but as indicated above it has been designed to allow for this to happen.

Modelling the interaction between people and landuse at the forest margin: the FLORES project

The Mafungautsi State Forest is located in central Zimbabwe, near the town of Gokwe. Considerable numbers of people live next to the forest, and there is considerable pressure to exploit the forest resources: primarily fuel wood, poles for construction, and grass for thatch in the vlei (annually-flooded river beds).

Acronyms

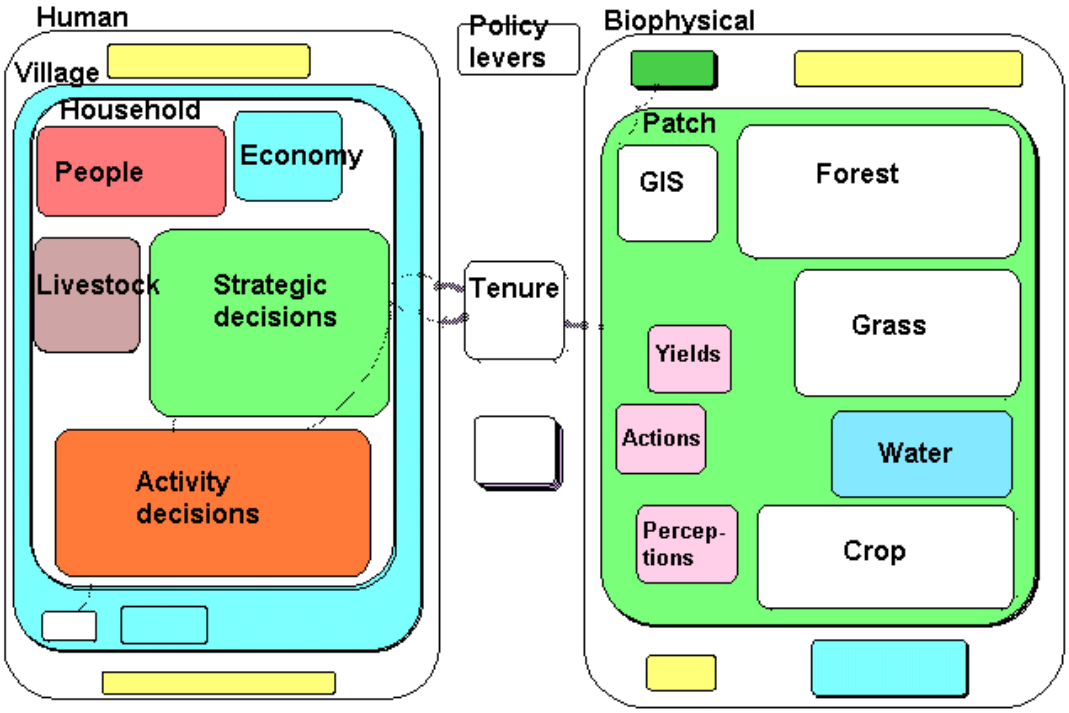
ACM	the CIFOR-led Adaptive Co-Managament project
CGIAR	Consulatative Group for International Agricultural Research, an umbrella organisation for a number of international institutes concerned with agriculture and natural resources.
CIFOR	the Centre for International Forestry Research, a CGIAR centre
FLAC	The FLORES Local Adaptation and Calibration package, which aims to enable local groups to make their own FLORES models.
FLORES	Forest Land-Oriented Resource Envisioning System, an international programme concerned with modelling the interaction between people and their land resources at the forest margin.

CIFOR has been undertaking research in the area for a number of years, concentrating on the relationship between the people, the forest resources, and the Forestry Commission, as part of the ACM project. They decided to take a lead role in developing a FLORES model for this context, based on a several villages on the north-west margin of the forest for which they had socio-economic data. The model was based on an earlier version of the FLAC training model, adapted by the addition of crop and water submodels on the biophysical side, and the development of a novel strategic decision-making submodel. The process of adapting the training model was undertaken with outside assistance, primarily in 4 FLAC workshops held between Sept 2000 and February 2001, in response to design requirements specified locally. The model was further refined in April 2001 as a demonstrator for the World Bank Rural Week meeting, held in Washington DC, 23-26th April 2001.

> *"This is unpublished work in progress by a team of researchers under CIFOR's ACM project in Zimbabwe. Please do not cite or use without permission from the authors (Haggith, Mudavanhu, Prabhu [r.prabhu@cgiar.org], Taylor, Muetzelfeldt, Vanclay, Sinclair, Matose, Nyirenda, Mapedza & Mutimukuru). This notice will be updated with a citable reference once the work has been published. Support for this work comes from the Department for International Development (U.K.) and the European Union."*

Structure

The overall structure of the model is shown in the following submodel diagram, which was taken directly from the Simile implementation of the model.



The model conforms closely to the 'standard' FLORES model, with a left-hand **Human** component and a right-hand **Biophysical** component. Inside the human submodel three **villages** are represented, with each village containing a variable number of **households**. The number of households increases when a family moves into the area, and decreases when a household dissolves because of, for example, the death of the only male adult. The biophysical side is represented in terms of a large number (900+) patches, based on the digitisation as polygons of the field boundaries from an aerial photograph.

The dynamics and behaviour of each household is modelled in 5 submodels:

- The submodel **People** models the changing population dynamics within the household, as children are born and household members die. It does this using a compartment-flow approach for each of 4 classes: children, adult males, adult females and elderly people.
- The submodel **Economy** models the economic status of the household, in terms of a single currency ('dosh': daily ordinary subsistence per household), which puts all household resources on a single basis.
- The submodel **Livestock** represents the dynamics of the household's herd of livestock. It is placed here, rather than in the Biophysical submodel, since each household has its own herd.
- The submodel **Strategic decisions** is responsible for determining each household's lifestyle strategy. This is represented in terms of the relative priority given by the household over the coming year to each of 7 activities (such as growing maize, growing cotton, or pole collection).
- The submodel **Activity decisions** is responsible for determining the household's week-by-week decisions. These primarily relate to the allocation of the household's labour resources to various activities, but include non-labour decisions, such as choice of crop to plant and allocation of cattle to grazing land.

The dynamics on the biophysical side is modelled in terms of 4 components:

- The **Forest** submodel calculates the dynamics of the miombo woodland in the State forest land. It is based on a model developed prior to FLORES in a workshop held in Zimbabwe, and represents the forest in terms of the number of trees in each of 4 classes: gullivers, poles, medium trees and large trees. It includes the reversion of trees to earlier classes if there has been a fire.
- The **Grass** submodel is also based on one developed in the same workshop. Its main relevance is in providing grass for thatch, but it also has a role in cattle grazing.
- The **Water** submodel is used to influence the growth of grass, since occasional drought years have a major influence on the people and their livelihoods.
- The **Crop** submodel is rather special. Three types of crop are allowed for: a food crop (maize); a cash crop (cotton); and vegetables (home garden). Any one patch can have only one of these, and the allocation of a crop to a patch is determined in the strategic decision-making submodel of the household that owns the patch.

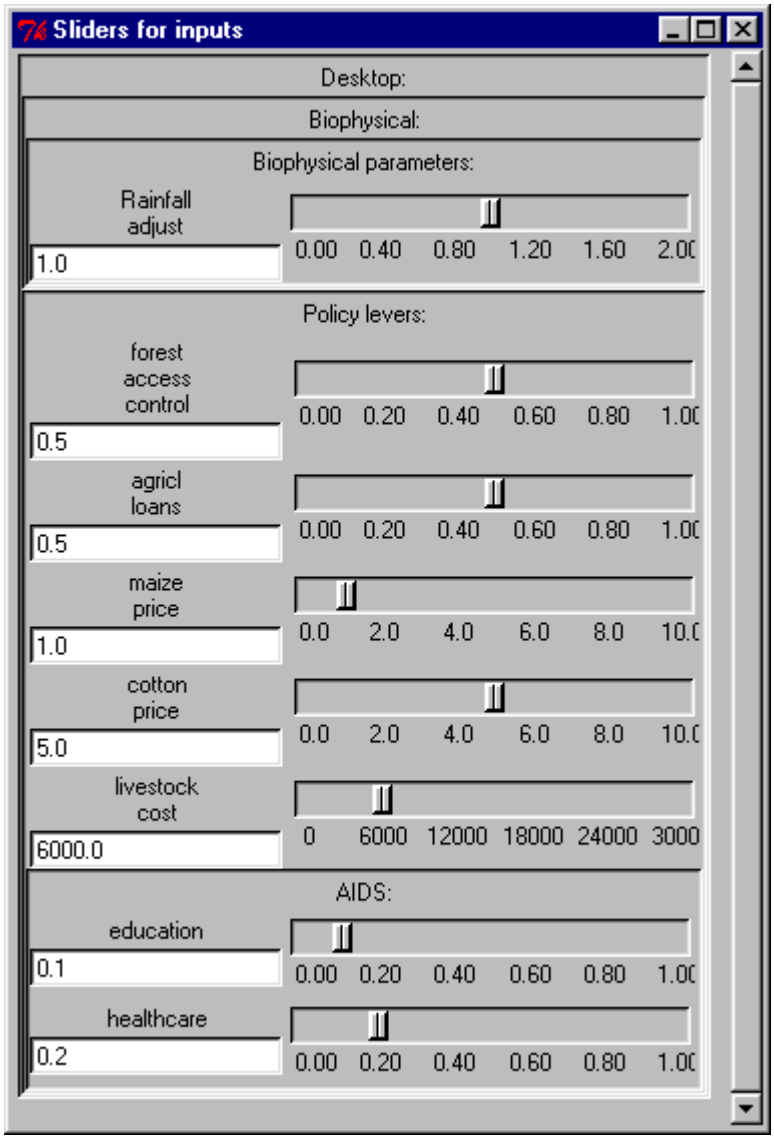
The GIS submodel contains fixed patch-level attributes (e.g. area, soil type), as well as the boundary coordinates for display purposes.

Households and patches are joined by a **Tenure** association. This allows for two types of tenure: ownership of a patch by a household; and access to common land. The main role of the tenure association is to channel information flows from patch to household and vice versa. Three types of information are transferred:

- **Perceptions** carry information about the patches they own to the households that own them.
- **Actions** carry information on the households weekly decisions from household to the patches they own.
- **Yields** carry information on the amount of produce harvested from patches to the households that own them

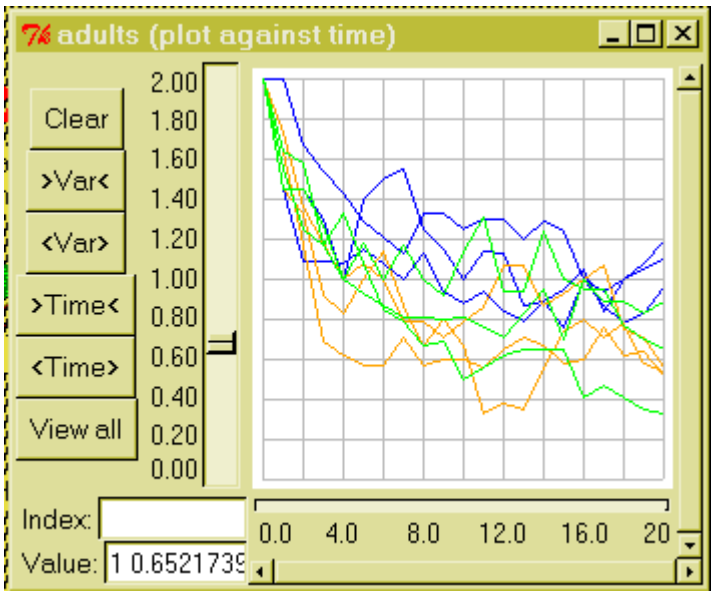
Using the model

The model is set up with a Control Panel that enables you to set various policy options. You simply move the sliders and continue running or restart the model .

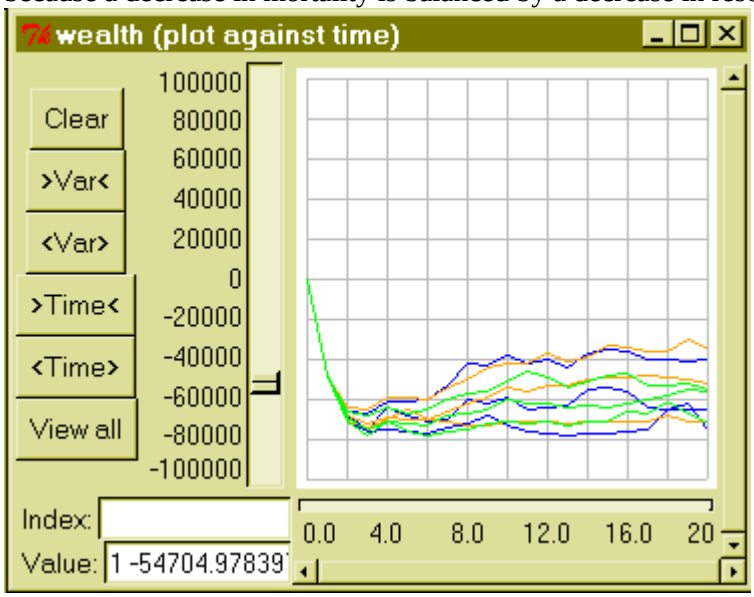


The behaviour of the model can be displayed using a number of display tools, including graphs and maps. The following diagrams are for illustrative purposes.

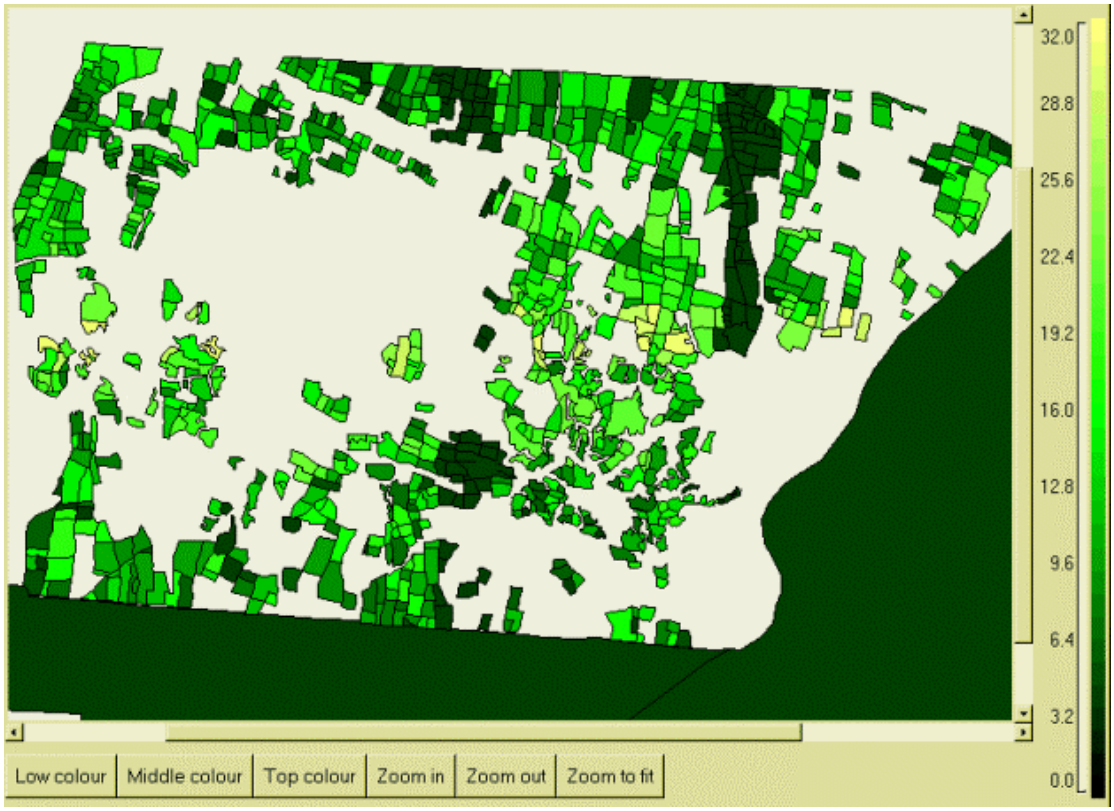
The two graphs below show how the model can be used to investigate how the community responds to changes in healthcare. The first shows the changes in the average number of adults per household in each of three villages. The orange lines show the standard run of the model. The blue lines reflect the effect of improved healthcare and health education: there is a slight improvement in the average household size. The green lines are for an increased incidence of AIDS: not a policy lever as such, but still something that the user of the model might wish to explore.



The following graph is for the same three cases, but shows wealth per household (in each of the 3 villages). No obvious effect of the healthcare/education policy is apparent, probably because a decrease in mortality is balanced by a decrease in resources per household.



The following map shows how indicators of performance can be displayed spatially. The solid dark area is part of the Mafungautsi State Forest. The small polygons are fields in the Batanai village: at the time this was prepared, only about half of the field boundaries had been digitised. The colour shows variations in crop productivity, with lighter areas indicating higher crop productivity running from left to right through the centre, with lower productivity on the higher (hence drier) land above and below this belt.


32.0
28.8
25.6
22.4
19.2
16.0
12.8
9.6
6.4
3.2
0.0
Low colour
Middle colour
Top colour
Zoom in
Zoom out
Zoom to fit