Comparing Multi-Agent Models Composed from Micro-Behaviours

Ken Kahn

Oxford University Computing Services

13 Banbury Road

Oxford, OX2 6NN, UK

kenneth.kahn@oucs.ox.ac.uk

Abstract

The behaviour of agents in a multi-agent model can typically be broken down into smaller independent units that we call *micro-behaviours*. The task of building a model can be seen as composing and parameterising micro-behaviours. Ideally the micro-behaviours can be drawn from a large library, but new micro-behaviours can be created and added as needed.

Initial experience with an implementation of this approach is that many micro-behaviours are shared across a diverse collection of models. Typically, models require some idiosyncratic micro-behaviours that are combined with more generic micro-behaviours.

Models built from micro-behaviours can be compared and contrasted by focussing on those micro-behaviours that differ. Micro-behaviours provide a higher-level way to compare the programs underlying different models. In addition to aiding model understanding and analysis, this approach facilitates the creation of models as well. We illustrate the creation of a series of models by composition of micro-behaviours using the so-called "stupid model" as a case study.

Keywords: Multi-agent modelling, micro-behaviours, BehaviourComposer, NetLogo, StupidModel

1. Introduction

The Constructing2Learn Project at Oxford University (Kahn 2007) is building a modelling tool called the *BehaviourComposer*. The BehaviourComposer has a web browser component for browsing web sites of code fragments we call *micro-behaviours*. These are bits of code that were carefully designed to be easily composed, parameterised, and understood. The BehaviourComposer user attaches these micro-behaviours to prototype agents. Typically, micro-behaviours for making copies are added to the prototype agents in order to create models containing many agents. When the user wishes to run the current model, the BehaviourComposer assembles a complete program and

launches it. The current prototype assembles NetLogo (Wilensky 1999) programs, but the framework could be adapted for other modelling systems such as Repast (North, Collier, and Vos 2006).

The current prototype uses a library of generic micro-behaviours organised into categories for initial position and state, movement, appearance, attribute maintenance, reproduction, death, and social networks. In addition there are micro-behaviours for creating graphs, sliders, buttons, and event logs. We are currently taking a variety of published models in life and social sciences and re-implementing them as collections of micro-behaviours. For example, we re-implemented a relative agreement model (Deffuant, Amblard, Weisbuch, and Faure 2002) as a collection of seven micro-behaviours, re-implemented a model of collective decision making (Couzin, Krause, Franks, and Levin 2005) using nine micro-behaviours, and the Sugarscape models (Epstein and Axtell 1996, Chapter 2) can be re-constructed using about forty micro-behaviours.

A major technical challenge is to design and build micro-behaviours so that they can be combined without concern for their order of execution. Each micro-behaviour is modelled as an autonomous process. A fish in a school, for example, may be concurrently running processes for avoiding fish that are too close, for aligning its orientation with neighbouring fish, for staying close to neighbouring fish, and for heading in a desired direction, as well as processes for modelling noise. These processes combine to generate the desired agent behaviour. Conflicts between these processes are avoided by careful use of scheduling routines (added to NetLogo) and attribute updating.

Micro-behaviours should not be confused with the software engineering concept of modules, components, or other programming language abstractions such as packages, classes, methods, or procedures. These modular constructs have interfaces that must be carefully matched in order to combine modules. They represent program fragments that run only if another fragment invokes them. Micro-behaviours run as independent processes or threads. They are designed to run simultaneously with a minimum (and in some cases zero) need to coordinate their execution order and interactions. Micro-behaviours resemble the structured processes in the LO programming language (Andreoli and Pareschi 1990).

The primary focus in building the BehaviourComposer is in educational tools for multi-agent model building. Students can quickly build, run, and analyse models without first mastering a programming language. We believe that the extremely modular model construction method means that the BehaviourComposer is also well-suited for comparing and contrasting models.

2.   Parameterisable and composable micro-behaviours

2.1.   A Simple Example Micro-behaviour

Each micro-behaviour is presented as a web page which can be accessed via links or a search engine just like any other web page. A section of the page is the program fragment itself. A button is automatically generated when the page is loaded in the BehaviourComposer's web component. When the button is pushed the code fragment is

added to the current prototype agent. The rest of the page by convention includes sections that

- describe the behaviour
- describe how to edit the micro-behaviour to produce variants
- provide links to related micro-behaviours
- describe how the program fragment implements the desired behaviour
- a history of edits to the micro-behaviour

Some pages also have references to published papers and links to sample models using the behaviour. The addition of formal specifications of micro-behaviours is a topic of future research.
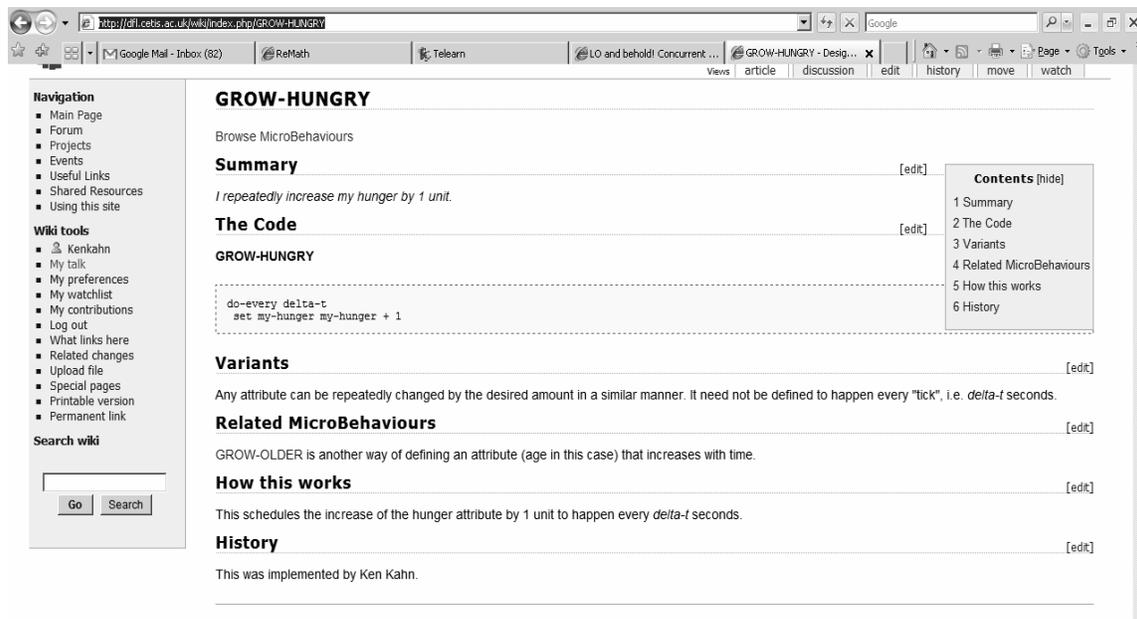
Figure 1. The Web Page of a Simple Micro-Behaviour

In Figure 1 we see the web page for a simple micro-behaviour for growing hungry. The code itself

```
do-every delta-t
    set my-hunger my-hunger + 1
```

is a NetLogo program extended with a scheduling primitive, *do-every*, described in the next section. The following section describes the NetLogo extensions we implemented to create and update attributes (*my-hunger* in this example).

2.2.   Scheduling Events

Code fragments defining micro-behaviours are ordinary NetLogo code enhanced with a scheduler. The system maintains a schedule for each agent. The schedule is specified using these primitives where *action* can be any NetLogo code:

- *do-at-setup* <action> performs *action* when the simulation is initialised

- *do-now* <action> performs *action* now

- *do-at* <time> <action> performs *action* when the clock has reached *time*

- *do-after* <interval> <action> performs *action* after *interval* time units

- *do-every* <interval> <action> performs *action* now and every *interval* time units

- *do-with-probability* <odds> <actions> performs *actions* with probability *odds*

- *do-repeatedly* <count> <actions> performs *actions count* times (if *count* is a non-integer then the *actions* may be performed an additional time where the odds are the fractional part of *count*)

- *do-if* <condition> <actions> performs *actions* if *condition* is true

- *when* <condition> <action> performs *action* once as soon as *condition* holds

- *whenever* <condition> <action> performs *action* whenever *condition* holds

In some cases it is possible to observe an animation of the execution of a model or the graphing of some aspects of the state of the model in *real-time*. The units for the scheduler are optionally interpreted as seconds and if the simulation is running faster than the schedule then the system will wait in order to reproduce a smooth and temporally accurate playback. Otherwise the simulation proceeds as normal but there may not be a constant ratio between simulation time and real-time due to varying or excessive computational demands.

2.3.   Creating and Maintaining Attributes

Most programming languages, including NetLogo, provide a means of creating object attributes and performing *immediate* updates of the values of attributes. Immediate updates of attributes introduce execution ordering dependencies. Consider, for example, two agents that update their position when they are within a maximum distance. If one agent updates its position, then the other will see the updated position and not the position the other agent had at the start of this round of activity. The model execution will then depend upon which agent or which activity ran first.

The BehaviourComposer is based upon the premise that models should ideally be defined by unordered collections of micro-behaviours. To enable this, *simultaneous* updates are supported. It is easy to express the requirement that all updates of state take place as if they were performed at the same instant.

NetLogo, like many programming languages, expects agent attributes ("breed variables" in NetLogo parlance) to be declared before use. The BehaviourComposer automates this so that any attribute whose name begins with *my-* becomes a breed variable without the need for a declaration. When an attribute needs to be both updated and read then the current and next value can be kept separate by using the *my-next-* form. After all actions scheduled for time *t* have completed, all attributes whose name begins with *my-* are set to the current value of the *my-next-* version of the attribute if it was updated. Predicates in conditionals can refer to the current state of an attribute by using the *my-* version of a variable, while code that updates a variable can use the *my-next-* version. In this way, execution order dependencies are eliminated.

For example, agents with the following micro-behaviour will at time *t+1* move to the left if that location was unoccupied at time *t*.

```
do-every delta-t
    let step-to-the-left my-x - 1
    if not any?
            all-individuals with [my-x = step-to-the-left]
        [set my-next-x step-to-the-left]
```

In contrast, agents with the following micro-behaviour will at time *t+1* move to the left if that location was unoccupied at time *t* by agents yet to run *and* unoccupied at time *t+1* by those agents that have already run.

```
do-every delta-t
    let step-to-the-left my-x - 1
    if not any?
            all-individuals with [my-x = step-to-the-left]
        [set my-x step-to-the-left]
```

If each agent in a line ran the first micro-behaviour only the leftmost agent would move at time 0, then the two leftmost agents at time 0, and so on. If, instead, they ran the second micro-behaviour then the same sequence of events *may* happen or they may all move left or many other possible outcomes can result from different execution orders. Immediate updates are simplest to implement and are the most common in modelling. Their idiosyncratic semantics (a mixture of time states) perhaps makes them less desirable than the simple semantics of simultaneous updates.

3.   Model Composition and Execution

Figure 2 illustrates the re-creation of the relative agreement model (Deffuant, Amblard, Weisbuch, and Faure 2002) in the BehaviourComposer. The bottom half is the browser component used to find and customise appropriate micro-behaviours. The code is displayed in an editable text area. Underneath is a button for adding the code to the currently selected agent prototype. The top half of the display contains the micro-behaviours currently associated with the agent prototype.
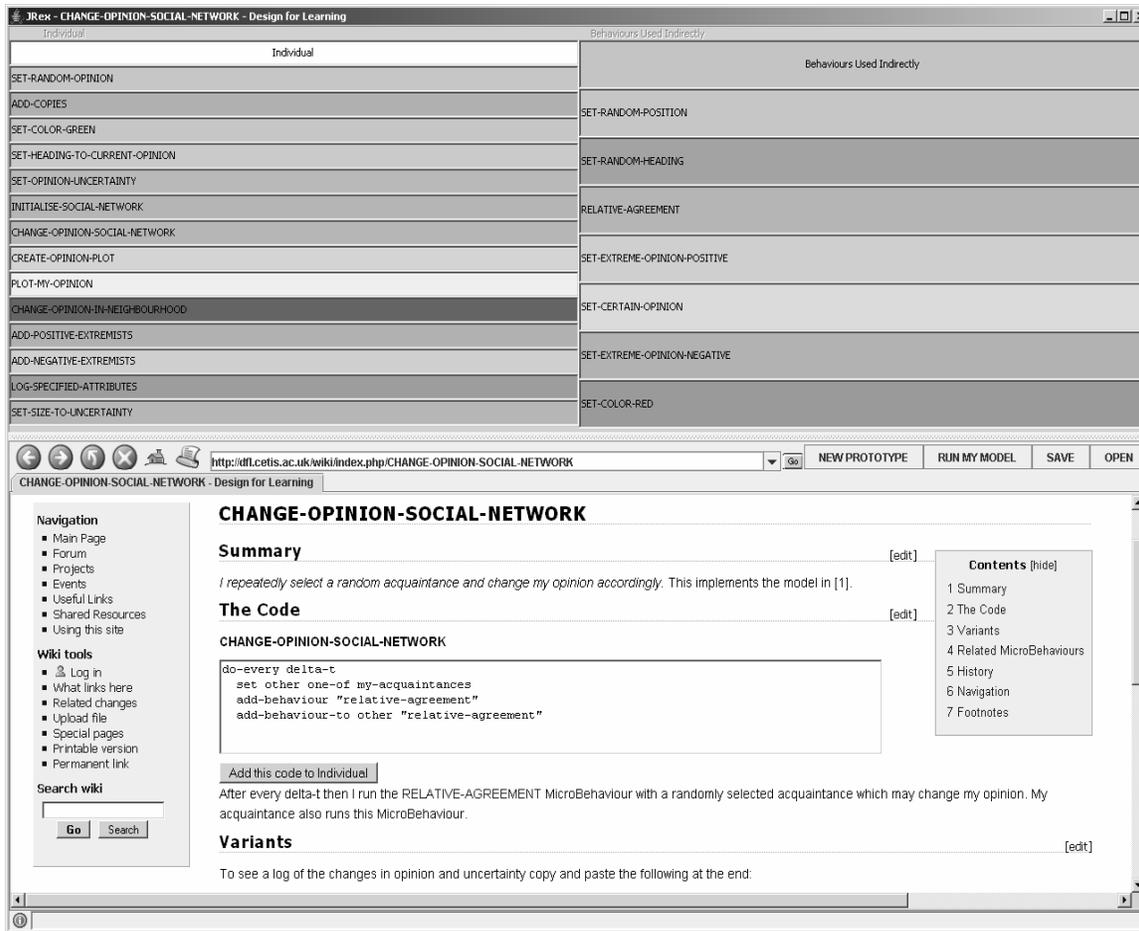
Figure 2. Relative Agreement Model in the BehaviourComposer

In Figure 3 we see a screen shot from the execution of this model in NetLogo. In addition to assembling the code, the BehaviourComposer added *setup* and *go* buttons and a graph of the average heading to the interface. Within the model each agent holds an opinion with some uncertainty. The opinion and uncertainty are modelled as numbers. In order to observe the simulation as it runs we map these numbers to attributes of the representations of agents on the display. The opinion of an agent is mapped to its heading (facing left is one extreme opinion and facing right is the other). The certainty with which an agent holds the opinion is mapped to the size of the agent. In order to visually distinguish the initial agents with extreme opinions and low uncertainty they are displayed as red.

Several variants of the model have been constructed by selecting different micro-behaviours for the pairing of agents for interaction. In some tests any two random agents can interact, in another the probability of interacting is a function of the spatial distance between them, and a third experiment permits interactions only within a social network. For example, in Figure 2 the micro-behaviour CHANGE-OPINION-IN-NEIGHBOURHOOD is inactivated while CHANGE-OPINION-SOCIAL-NETWORK is active. They can be switched with two menu selections and the model can be re-run with a different scheme for how agents encounter one another.
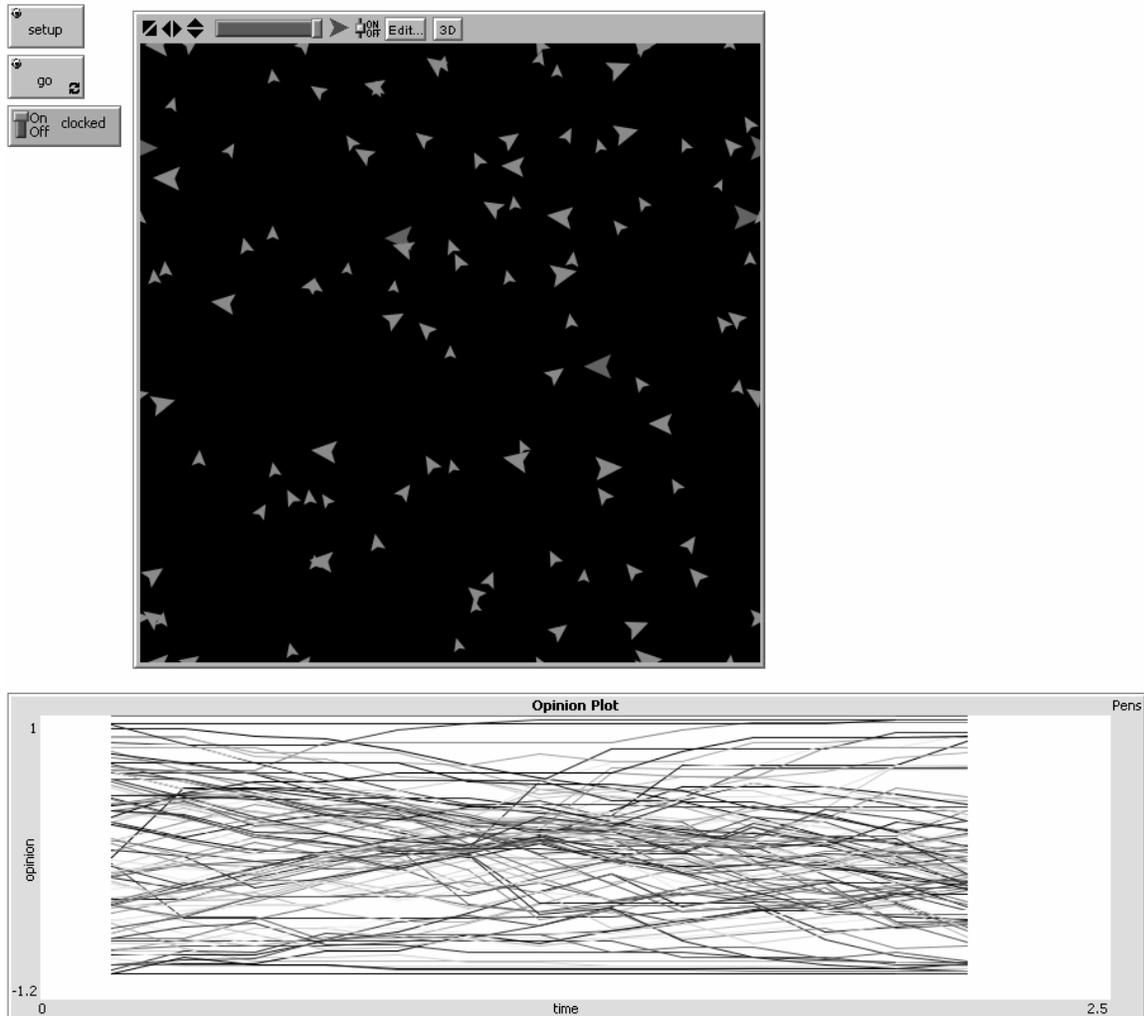
Figure 3. The BehaviourComposer model of relative agreement in NetLogo

4.   A Detailed Comparison of Models

While the BehaviourComposer has been used to construct several models in the social sciences, we feel it is more instructive to compare a model created in the BehaviourComposer with the so-called "stupid model" of predators and prey that has been implemented in at least five different modelling tools (Railsback, Lytinen, and Jackson 2006). The issues that arise are not limited to ecological modelling but any model with mobile interacting and reproducing agents in an environment. There is a progression of 16 increasingly more complex "stupid models" but here we focus upon the final model.

The model consists of bugs, predators, and food cells. Bugs wander around, eat, grow, reproduce, and die. Predators move and eat bugs. Cells grow food. The attributes of the cells are defined by a data file. Some model parameters are controllable via the user interface. The model displays the dynamic state, a histogram of bug sizes, and a graph of the bug population. A log file of some simple statistics is produced. A screenshot of the BehaviourComposer implementation of the model can be seen in Figure 4.
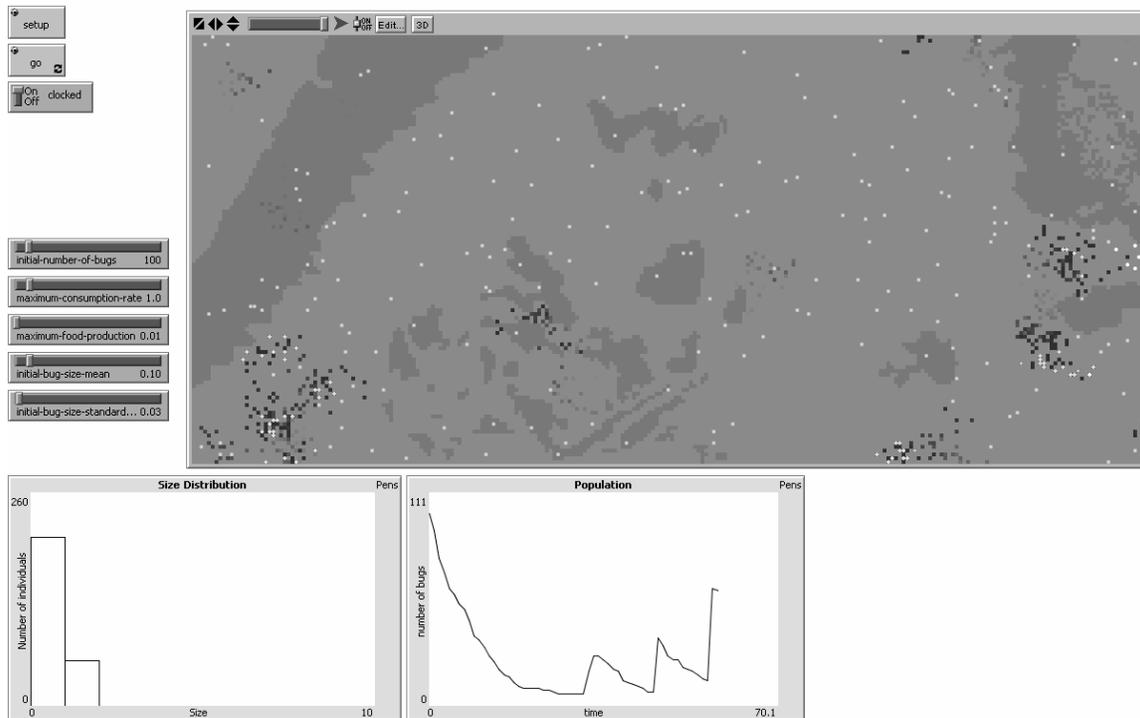
Figure 4. A snapshot from running the Stupid Model

4.1.   Movement to a nearby unoccupied location

In the first version of the stupid model the bugs are defined to move as follows:

> 100 bug agents are created. They have one behaviour: moving to a randomly chosen grid location within +/- 4 cells of their current location, in both the X and Y directions. If there already is a bug at the location (including the moving bug itself—bugs are not allowed to stay at their current location unless none of the neighbourhood cells are vacant), then another new location is chosen. This action is executed once per time step. (Railsback, Lytinen, and Grimm 2005)

It is interesting to note that none of the five implementations of this model implemented this correctly. They all implemented code that repeatedly tried random neighbourhood cells until one is found to be vacant. (The source code of only one implementation had a comment that mentions that this will loop forever if the entire neighbourhood is full.) Because the space contains 10,000 cells and there are only 100 bugs the odds are very slim that a 9x9 region will fill. Two micro-behaviours were implemented: (1) a correct implementation that selects a random member of the list of unoccupied neighbouring cells or stays put if this is an empty list and (2) the incorrect (but much faster) repeated selection of a random location followed by a test of whether it is empty. The bug agents can easily be associated with either micro-behaviour. Perhaps the ideal solution is to try a number of random locations and then if no free cells have been found to switch to the slower but safer version.

4.2.   Optimal movement

> In its move method, a bug identifies a list of all cells that are within a distance of 4 grids but do not have another bug in them. (The bug's current cell is included on this list.) The bug iterates over the list and identifies the cell with highest food availability. The bug then moves to that cell. (Railsback, Lytinen, and Grimm 2005)

The eleventh version replaced movement to a nearby free random location by movement to the nearby free cell with the most food available. The NetLogo model produced by the BehaviourComposer and the hand-crafted NetLogo model should have differed only in small details due to scheduling. Visually the executions looked similar but when comparing the logs of the minimum, maximum, and mean size of the bugs there was a significant difference in the minimum and mean values. This difference was due to the micro-behaviour's use of simultaneous update of location rather than the NetLogo model's immediate update. Changing the micro-behaviour to update *my-x* and *my-y* rather than *my-next-x* and *my-next-y* fixed this.

This micro-behaviour bug illuminates the oddness of *both* interpretations of movement. The micro-behaviour was modelling bugs that simultaneously determined their optimal cell and then each moved to it. This doesn't respect the specification that a location can have at most only one individual. The alternative which was implemented by all the others has an odd notion of time. It is as if one bug moves to its optimal location and only *then* does the next bug move and so on. Conceptually it takes *n* simulation time steps for *n* bugs to move. Note that the specification is silent on this issue but the intention was probably the sequential movement of each bug.

4.3.   Ambiguous specification of predator behaviour

> Predators have one method: hunt. First, a predator looks through a shuffled list of its immediately neighboring cells (including its own cell). As soon as the predator finds a bug in one of these cells it "kills" the bug and moves into the cell. (However, if the cell already contains a predator, the hunting predator simply quits and remains at its current location.) If these cells contain no bugs, the predator moves randomly to one of them. (Railsback, Lytinen, and Grimm 2005)

The rule about the cell already containing a predator is ambiguous. It could mean the cell that it finds containing a bug or it could be the next cell in the shuffled list. The first interpretation is biologically plausible while the second one is the one the other implementations chose. Note that it leads to pairs of predators remaining immobile if there is no prey in their neighbourhood. Both micro-behaviours were created.

4.4.   Ambiguous specification of prey reproduction

> New bugs are placed at the first empty location randomly selected within +/- 3 cells of their parent's last location. If no location is identified within 5 random draws, then the new bug dies. (Railsback, Lytinen, and Grimm 2005

The ambiguity here is whether the 5 random draws should be interpreted as per new bug or per reproduction. The Mason, RePast, and BehaviourComposer interpreted it as per

bug while the NetLogo model implemented the limit per reproduction event. The two Swarm implementations ignored this completely and hence could loop forever if the population became too dense and no nearby unoccupied locations could be found.

4.5.   Many details wrong

The handcrafted NetLogo model differs from the specification and other models in many details. These differences probably do not lead to significantly different outcomes but should be eliminated in the spirit of doing careful model-to-model comparisons. These include:

- The neighbourhood of a bug is defined as circular rather than square. (This is fixed in versions 15 and 16 when the code was modified to deal with the change from a torus to a bounded plane.)

- Beginning with version 15 food production should be table-driven rather than random. The NetLogo model instead uses the table as the maximum random value.

- The earlier versions of the model are supposed to stop after the largest bug reaches 100 units. NetLogo continues until it reaches 1000.

- Later versions of the model were to stop after 1000 steps while the NetLogo model does not.

- The patches are to be coloured from black to green depending upon how much food is available. The NetLogo version colours from black to white via green instead.

- The histogram should have ten bars not one hundred.

4.6.   The BehaviourComposer model

The bug behaviours were broken down into 12 micro-behaviours (and 7 variant micro-behaviours of earlier versions). The behaviour of predators is defined by 4 micro-behaviours. The world or environment where this takes place is defined by 6 micro-behaviours. And the observer which consists of a histogram, a graph, and 5 sliders for controlling model parameters was defined by 11 micro-behaviours. Figure 5 is a display of the entire model (including 9 micro-behaviours for alternative behaviours that were needed in earlier versions but have been inactivated in version 16).
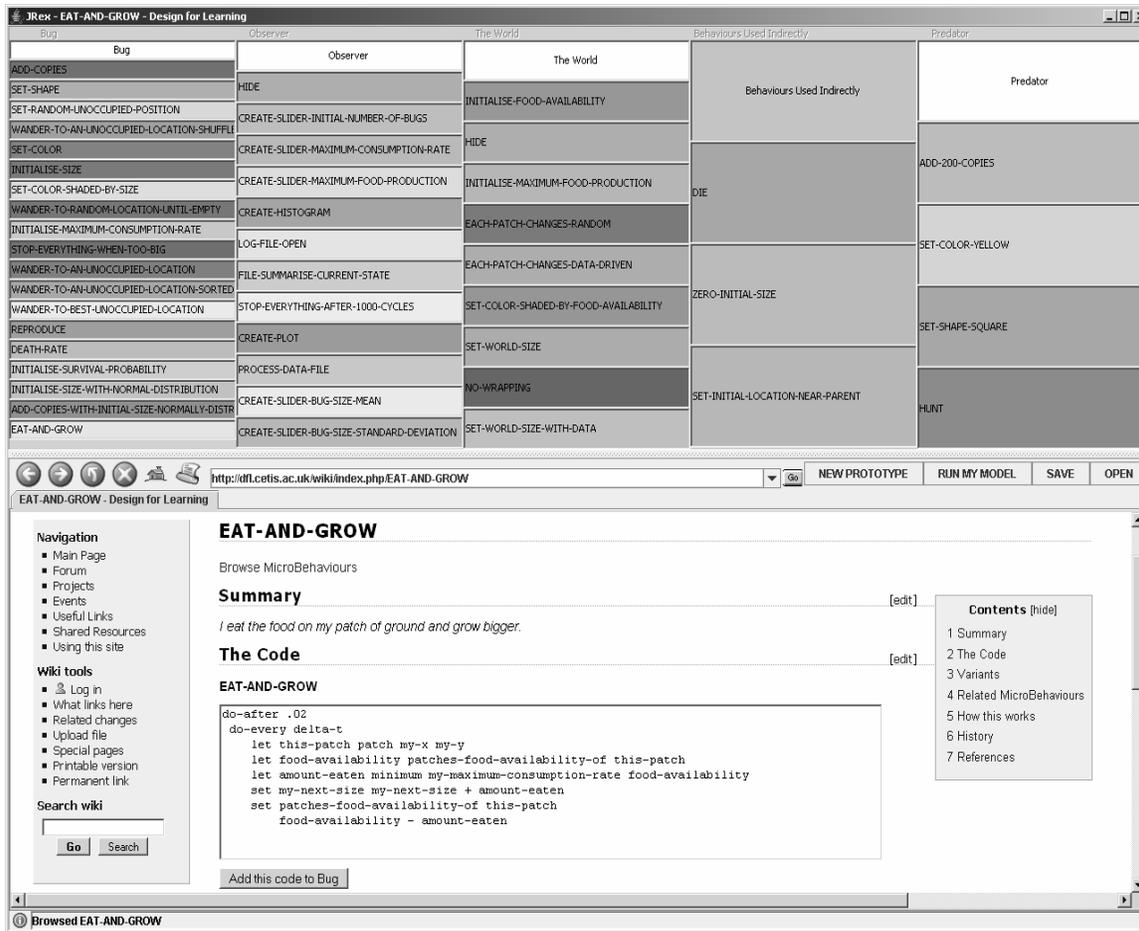
Figure 5. The final Stupid Model in the BehaviourComposer

While we were able to break the model into small modular pieces the specification of the problem runs counter to the strength of the BehaviourComposer. The BehaviourComposer was designed to support truly concurrent activities. The Stupid Model strongly constrains the order in which actions occur. It specifies that individuals in populations take turns acting rather than running concurrently.

To accommodate these ordering constraints the BehaviourComposer's scheduler was heavily used. Each kind of action was given different delays so that some actions were delayed 0.01 time units, others 0.02, and some 0.03 to ensure they occur in the specified order. Version 10 introduced the constraint that bugs act in order of their size. To express this each bug's schedule was shifted by *.001 / (size + 1)* so that the larger bugs were delayed the least. (Since *size* can be 0 we added 1.) Note that the alternative solution of sorting the list of bugs by size is not supported by the BehaviourComposer. Explicit manipulation of lists of different kinds of agents runs counter to the pedagogic goals underlying the design of the BehaviourComposer. The mechanism of the top-level loop is opaque.

The scheduling constraints of the Stupid Model were satisfied at the cost of clarity and ease of composition. Rather than having independent micro-behaviours their scheduling introduces dependencies. For example, adding a new action between one delayed by 0.01

time units and another delayed by 0.02 requires that one delays the new by a value in between. This introduces a dependency upon the schedule of the others and any future additions. It is much simpler to build and reason about a model that relies upon simultaneous updates to become insensitive to the order of execution.

5. Conclusion

Even when the same programming language and libraries are used, it is not easy to compare and contrast models (Hales, Rouchier, and Edmonds 2003). Well-structured programs are modular but cannot be compared as easily as programs constructed from micro-behaviours. Micro-behaviours are independent concurrent processes, and hence one can compare the behaviours of agents in different models by looking for the micro-behaviours they have in common. One can then focus attention upon the sets of micro-behaviours that are different.

Well-designed and tested micro-behaviours can become *de facto* standard building blocks. As the library of well-established micro-behaviours grows we expect it will aid both in model construction and in analysis. Furthermore, we entertain the possibility that semantically equivalent micro-behaviours will be implemented for different programming and modelling environments, thereby facilitating cross-language model comparisons.

Any NetLogo model can be constructed via the BehaviourComposer. The big open question is: how general is the micro-behaviour approach? Can most models be broken down into concurrently executing components? Or will they require large monolithic parts? How often are micro-behaviours reusable in other models? When is it more effective to simply build the model directly in NetLogo? Only additional research will answer these questions.

6. Acknowledgements

7. References

ANDREOLI, J. and PARESCHI, R. (1990) LO and behold! Concurrent structured processes, *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications* (Ottawa, Canada). OOPSLA/ECOOP '90. ACM Press. Also published in *ACM SIGPLAN Notices*, Volume 25, Issue 10 Oct. 1990

COUZIN, I.D., KRAUSE, J., FRANKS, N.R. & LEVIN, S.A.(2005) Effective leadership and decision making in animal groups on the move, *Nature* 433, 513-516.

DEFFUANT, G., AMBLARD, F., WEISBUCH, G. and FAURE, T. (2002) How can extremism prevail? A study based on the relative agreement interaction model, *Journal of Artificial Societies and Social Simulation*, vol. 5, no. 4, http://jasss.soc.surrey.ac.uk/5/4/1.html

EPSTEIN J. and AXTELL R. (1996) *Growing Artificial Societies,* Brookings Institute Press, The MIT Press.

HALES, D., ROUCHIER, J. and EDMONDS, B. (2003) Model-to-Model Analysis, *Journal of Artificial Societies and Social Simulation*, vol. 6, no. 4, http://jasss.soc.surrey.ac.uk/6/4/5.html

KAHN, K. (2007) Constructing2Learn Project Web Site, http://dfl.cetis.ac.uk/wiki/index.php/Constructing2Learn.

NORTH, M.J., COLLIER, N.T. and VOS, J.R. (2006) Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit, *ACM Transactions on Modeling and Computer Simulation*, Vol. 16, Issue 1, pp. 1-25, ACM, New York, New York, USA.

RAILSBACK, S. F., LYTINEN, S. L. and GRIMM, V. (2005) StupidModel and Extensions: A Template and Teaching Tool for Agent-based Modeling Platforms, http://condor.depaul.edu/~slytinen/abm/StupidModelFormulation.pdf

RAILSBACK, S. F., LYTINEN, S. L. and JACKSON, S. K. (2006) Agent-based simulation platforms: review and development recommendations. *Simulation* 82: 609-623.

WILENSKY, U. (1999) NetLogo. http://ccl.northwestern.edu/netlogo/ Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

8.   Availability of the models and the BehaviourComposer

A beta version of the BehaviourComposer is available for download from http://dfl.cetis.ac.uk/wiki/index.php/Beta_testing. The BehaviourComposer Stupid Model is available at http://dfl.cetis.ac.uk/wiki/index.php/Browse_MicroBehaviours.