# Building Computer Models from Small Pieces

**Ken Kahn**
**Oxford University**
kenneth.kahn@oucs.ox.ac.uk

**Keywords:** micro-behaviors, simulation construction kits, Behavior Composer, *NetLogo*

**Abstract**

Computer models can be built by assembling code fragments. Here we describe the *BehaviourComposer,* which supports browsing for small modular program pieces called *micro-behaviors* and their assembly and execution. Using a web browser, the model builder finds and customizes micro-behaviors and associates them with prototypical agents. These micro-behaviors run as independent processes. Different combinations of micro-behaviors produce the desired behavior of each element of the model.

One of the challenges is creating a runtime environment in which several un-ordered micro-behaviors can run together without conflict or the need to explicitly interface them. Another challenge is creating and organizing libraries of micro-behaviors.

## 1. INTRODUCTION

The Constructing2Learn Project at Oxford University [Kahn 2007b] is building a modeling tool called the *BehaviourComposer*. The *BehaviourComposer* has a web browser component for browsing web sites of code fragments called *micro-behaviors*. These are bits of code that were carefully designed to be easily understood, composed, and parameterized. The *BehaviourComposer* user attaches these micro-behaviors to prototype agents. In order to create models containing many instances of a prototype agent, a micro-behavior for making copies is added to the prototype. 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 modeling systems such as *Repast* [North et al 2006].

The current prototype uses a library of generic micro-behaviors organized into categories for initial position and state of model elements, movement, appearance, attribute maintenance, reproduction, death, and social networks. In addition there are micro-behaviors for creating graphs, histograms, sliders, buttons, and event logs. We are currently taking prototypical published models in zoology and social sciences and re-implementing them as collections of micro-behaviors. For example, we re-implemented a relative agreement model [Deffuant et al 2002] as a collection of seven micro-behaviors and re-implemented a model of vaccinations [Scherer and McLean 2002] using ten micro-behaviors.

A major technical challenge is to design and build micro-behaviors so that they can be combined without concern for their order of execution. Each micro-behavior is modeled 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 neighboring fish, for staying close to neighboring fish, and for heading in a desired direction, as well as processes for modeling noise. These processes combine to generate the desired agent behavior. Conflicts between these processes are avoided by careful use of scheduling routines and support for simultaneous updating of attributes (we added both to *NetLogo*).

Micro-behaviors 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 them. They represent program fragments that run only if another fragment invokes them. Micro-behaviors 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-behaviors 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 analyze models without first mastering a programming language. We are currently exploring whether the highly modular model construction method of the *BehaviourComposer* will also be well-suited for constructing models for research purposes. A novel economic model of network formation is being constructed using the *BehaviourComposer* for this purpose.

## 2. PARAMETERISABLE AND COMPOSABLE MICRO-BEHAVIORS

The building blocks of models constructed with the *BehaviourComposer* are micro-behaviors: small, coherent, and independent program fragments.

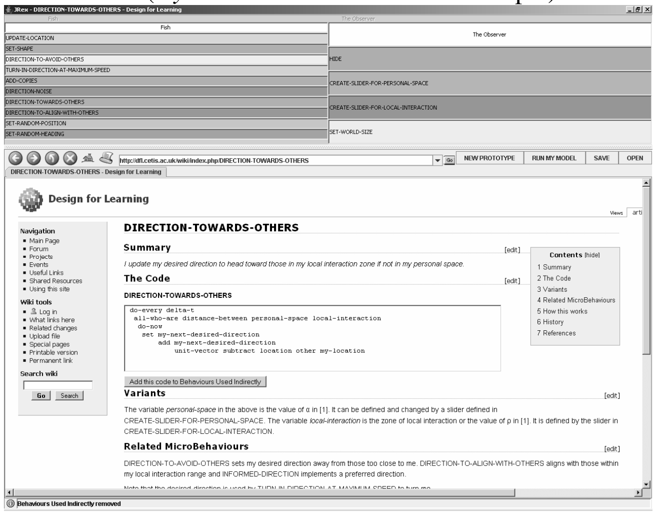
### 2.1. A Typical Micro-behavior

Each micro-behavior is presented as a web page which can be accessed via links, tags, 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. By convention, the rest of the page includes sections that

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

Some pages also have references to published papers and links to sample models using the behavior. The addition of formal specifications of micro-behaviors is a topic of future research.

In the Figure 1 we see the *BehaviourComposer* application displaying the model composition window and a web page for a micro-behavior for moving towards others. The code itself is a *NetLogo* program extended with a scheduling primitive, *do-every*, described below, and an iteration primitive *all-who-are*. The following sections describe the *NetLogo* extensions for scheduling and attribute maintenance (*my-desired-direction* in this example).



DIRECTION-TOWARDS-OTHERS

Summary

*I update my desired direction to head toward those in my local interaction zone if not in my personal space.*

The Code

DIRECTION-TOWARDS-OTHERS

```
do-every delta-t
 all-who-are distance-between personal-space local-interaction
  do-now
   set my-next-desired-direction
       add my-next-desired-direction
           unit-vector subtract location other my-location
```

Variants

The variable *personal-space* in the above is the value of α in [1]. It can be defined and changed by a slider defined in CREATE-SLIDER-FOR-PERSONAL-SPACE. The variable *local-interaction* is the zone of local interaction or the value of ρ in [1]. It is defined by the slider in CREATE-SLIDER-FOR-LOCAL-INTERACTION.

Related MicroBehaviours

DIRECTION-TO-AVOID-OTHERS sets my desired direction away from those too close to me. DIRECTION-TO-ALIGN-WITH-OTHERS aligns with those within my local interaction range and INFORMED-DIRECTION implements a preferred direction.

**Figure 1** – Typical *BehaviourComposer* Screen Shot

### 2.2. Scheduling Events

Code fragments defining micro-behaviors consist of ordinary *NetLogo* code enhanced with a scheduler. Our *NetLogo* extensions maintain a schedule for each agent. The schedule is specified using these primitives where *actions* can be any *NetLogo* code:

- *do-at-setup* <actions> performs *actions* when the simulation is initialized
- *do-now* <actions> performs *actions* immediately
- *do-at* <time> <actions> performs *actions* when the clock has reached *time*
- *do-after* <interval> <actions> performs *actions* after *interval* time units
- *do-every* <interval> <actions> performs *actions* 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> <actions> performs *actions* once as soon as *condition* holds
- *whenever* <condition> <actions> performs *actions* 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 real time, the system slows down in order to reproduce a smooth and temporally accurate playback. If the user runs the simulation "un-clocked" it proceeds as normal, but there will 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 specified 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. While the agents are conceptually running simultaneously, the state of the model will depend upon the order in which the simulation engine runs the agents. This is often undesirable.

The *BehaviourComposer* is based upon the premise that models should ideally be defined by unordered collections of micro-behaviors. To enable this, *simultaneous* updates are supported. This enables the model builder to express the requirement that all updates of state should 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 read and updated 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. 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 simultaneous update micro-behavior 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 immediate update version of this micro-behavior 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. It differs from the simultaneous update version in that the last line is

```
      [set my-x step-to-the-left]
```

If each agent in a line ran the simultaneous update micro-behavior only the leftmost agent would move at time 0, then the two leftmost agents at time 1, and so on. If they ran the immediate update micro-behavior then the same sequence of events *may* happen, or they may all move left: many other possible outcomes can result from different execution orders. Immediate updates are simplest to implement and are the most common in modeling. We believe their idiosyncratic semantics (a mixture of time states) makes them less desirable, in general, than the simple semantics of simultaneous updates. The choice between the two kinds of update can be made by the micro-behavior programmers on a case-by-case basis.

## 3. MODEL CONSTRUCTION

Here we illustrate a typical scenario in which a model is constructed using the *BehaviourComposer*. The model is based upon [Couzin et al 2005] which explores collective decision making in animal groups. A school of fish is modeled using micro-behaviors for avoidance, attraction, alignment, noise, informed direction, and maximum turning. Some of the scientific questions explored with this model are how the school behaves if a small fraction of the individuals tend to move in an "informed" direction. Will the entire school follow? What if there is more than one informed direction?

Using the *BehaviourComposer* one can browse to a page with links to about a dozen micro-behaviors specially constructed for these kinds of models. A good model construction heuristic is to build and test successively more complex models. Students can begin by modeling a single fish moving in a straight line at a constant velocity. We can provide (detailed or very open-ended) instructions to students to browse to a page with the UPDATE-POSITION micro-behavior whose code fragment is:

```
do-every delta-t
   go-forward 1 * delta-t
```

Following the advice on the page students replace the '1' with the desired speed. One could replace it with the variable name of a slider to be able to easily change the speed during testing. Students also add and customize the SET-SHAPE micro-behavior so our object looks like a fish.

Students now execute the model by clicking on the 'Run My Model' button. A complete *NetLogo* program is assembled, *NetLogo* is launched, and, by clicking on the 'Go' button, they see a fish move upwards.

Next, to make the fish avoid those too close, students add the DIRECTION-TO-AVOID-OTHERS micro-behavior:

```
do-every delta-t
   all-who-are distance-within 1
      do-now
         set my-next-desired-direction
            subtract
               my-next-desired-direction
               unit-vector
                  subtract
                     location other
                     my-location
```

We find that using the first person for the "owner" of a micro-behavior makes for more easily comprehensible descriptions of behaviors. We describe this code as "I consider all others that are at most one unit away from me. For each one I update my desired direction by subtracting the unit vector from it to me." Later we replace the '1' in the code by *personal-space*, a variable controlled by a slider.

The code above implements the first equation in [Couzin et al 2005]:

$$\mathrm{d}_i(t+\Delta t) = -\sum_{j\neq i}\frac{c_j(t)-c_i(t)}{\left|c_j(t)-c_i(t)\right|}$$

The model now needs the TURN-IN-DESIRED-DIRECTION-AT-MAXIMUM-SPEED micro-behavior to convert the desired direction into a new heading. We need to allow for noise to model sensory or motor inaccuracies (which can be zero) and to impose a maximum turning speed. The micro-behavior is defined as:

```
do-every delta-t
  let desired-heading
      direction-to-heading
         my-desired-direction +
         my-direction-noise
  let desired-delta-heading
      canonical-heading
```

```
        (desired-heading - my-heading)
  let maximum-turn 2 * radian * delta-t
  set my-next-heading
      my-next-heading +
      within-range desired-delta-heading
                   (- maximum-turn)
                   maximum-turn
  set my-next-desired-direction 0
```

There is a problem here with the *BehaviourComposer*'s inability to specify that this micro-behavior should run after all the other micro-behaviors that contribute to the value of the desired direction have completed. Currently, we address this by using the *BehaviourComposer*'s scheduler to postpone this micro-behavior by a tiny amount relative to the other micro-behaviors. By prefixing `do-after .001` to the code we specify that this code regularly runs .001 time units after the others.

To test the avoidance behavior students need more than one fish. They can add the ADD-COPIES micro-behavior to the fish:

```
do-at-setup
   add-copies 1 []
```

This will create a single copy that has no additional micro-behaviors. When students run the model they will see two fish (the original prototype and the copy) at the same position move in unison. This is because the copy is an identical copy. They could resolve this by adding micro-behaviors to the call to `add-copies` as:

```
add-copies 1
   ["SET-RANDOM-POSITION"
    "SET-RANDOM-HEADING"]
```

Or they can add the DIRECTION-NOISE micro-behavior to model the inability of fish to exactly control their heading. With either solution, or both, they see their two fish moving and veering away from each when they are within one unit of each other.

We found it useful to change the '1' in the call to *add-copies* to a large number to test our model with many fish. Later when students add additional micro-behaviors we suggest that they change it back to '1' in order to test the model in the simplest situation and then restore the large number for realistic testing.

One issue with testing our model is the need to specify the geometry of the space these fish are swimming in. The default in the *BehaviourComposer* (and in *NetLogo*) is a torus. It is very convenient to have the fish appear on the side of the display when it swims off the opposite side. If we were to use a micro-behavior to set the geometry of the world to a 2D plane or a 3D volume then we would need to use a micro-behavior to initialize the position and heading of our fish to ensure that they approach each other for testing. Once we add a micro-behavior that causes fish to be attracted to each other then we could more easily test other geometries.

Our ideal student next adds the DIRECTION-TOWARDS-OTHERS micro-behavior to our fish:

```
do-every delta-t
  all-who-are
     distance-between personal-space
                      local-interaction
   do-now
     set my-next-desired-direction
         add my-next-desired-direction
             unit-vector
                 subtract location other
                 my-location
```

This differs from DIRECTION-TO-AVOID-OTHERS in that it adds rather than subtracts the unit vector from the other to "me" and has a different range of distances for which it applies. Note that due to the *BehaviourComposer*'s support for simultaneous updates the same desired direction will be computed regardless of the order of execution of DIRECTION-TOWARDS-OTHERS and DIRECTION-TO-AVOID-OTHERS. Similarly the DIRECTION-TO-ALIGN-WITH-OTHERS micro-behavior can be added and its execution can be interleaved with the others.

A slightly different model of fish behavior specifies that if there are any fish to avoid then the avoidance behavior takes precedence and the attraction and alignment behaviors do not occur. This slightly interferes with the pure independence of the micro-behaviors. It can be implemented by setting a new attribute in DIRECTION-TO-AVOID-OTHERS to true and adding a condition to the other micro-behaviors that they don't run if the attribute is true.

A micro-behavior to give a fish an "informed direction" could be implemented as another process that adds or subtracts from the fish's desired direction. The published model instead introduces a weighting factor that is used to combine the unit vectors of the informed direction and the desired direction. Again this interferes with the strong independence of the micro-behaviors, since the INFORMED-DECISION micro-behavior must run after the others have computed the desired direction.

The model of the fish can be enhanced in various ways [Reynolds 1987] such as introducing a cone of vision so that a fish only interacts with those it can see.

We have also explored the construction of games by adding to the simulation individuals that are controlled interactively. A student can explore the behavior of the school of fish by controlling one or more fish, perhaps to learn first-hand the extent to which the school can be influenced by a proportionately small number of individuals.

In a similar manner micro-behaviors can be associated with the observer to obtain graphs, histograms, monitors, and statistics. Other micro-behaviors can be associated with the world to specify its scale, geometry, and the state of the environment.

## 4. STRENGTHS OF MICRO-BEHAVIORS

Micro-behaviors are organized into a web site where each micro-behavior has a page that includes much more than just the code fragment needed for execution. Students can acquire an understanding of what a micro-behavior does without reading the program code. Simple edits to the code are possible without programming expertise. Micro-behaviors are designed to be the smallest coherent unit of behavior and as such are often easy to understand. Micro-behaviors can often be better understood by executing them in isolation or with only a few accompanying micro-behaviors.

When appropriate micro-behaviors are available modeling becomes a "middle-out" activity of composing (upward) and editing (downward) micro-behaviors rather than the normal bottom-up programming activity.

A familiar web browser is used to search and browse for micro-behaviors. We use a Wiki to support the collaborative creation of libraries of micro-behaviors. Each page for a micro-behavior can thereby support discussions by users and authors.

The decomposition of a model into independent concurrent processes enables students to rebuild the model at a high level, focusing on domain issues rather than technical ones. A set of micro-behaviors can be composed in different ways to form a rich family of models. The simple example presented here includes micro-behaviors for avoidance, attraction, alignment, noise, and informed movement. Students can explore different subsets and different customizations of these micro-behaviors.

The support for the expression of simultaneous updates often enables micro-behaviors to be executed in any order. Indifference to the execution order enables the model builder to more easily construct and experiment with different models.

Another advantage of models built out of micro-behaviors is that they are easier to understand and compare than relatively monolithic program sources [Kahn 2007a].

## 5. WEAKNESSES OF MICRO-BEHAVIORS

The critical open question is: how often can worthwhile models be decomposed into micro-behaviors? We have built a small number of models other than the collective decision making model reported here. They include a model of the spread of disease, another for modeling the spread of extremist opinions, a predator and prey model, the SugarScape model [Epstein and Axtell 1996], and an economic model of network formation. Our experience has been that most models are "nearly decomposable" [Simon 1962]. Dealing with the weak or occasional interactions between micro-behaviors does introduce complexity and dependencies that reduce the benefits of our approach.

There are also issues of execution speed and memory usage. In order to achieve modularity some computations are repeated. For example, both the DIRECTION-TOWARDS-OTHERS and DIRECTION-TO-ALIGN-WITH-OTHERS micro-behaviors compute the set of individuals whose distance is within a specified range. The use of the scheduler to impose ordering constraints also entails some overhead. An open question is whether an optimizing implementation could eliminate these kinds of additional costs.

## 6. FUTURE RESEARCH

The *BehaviourComposer* is a proof-of-concept prototype. There are many ways of enhancing it, including a drag-and-drop user interface, support for hierarchical grouping of micro-behaviors, a better way to customize micro-behavior pages, and support for import and export to model repositories. Hierarchical grouping together with the support for prototypes should provide comparable functionality to class inheritance in conventional object-oriented systems. The libraries of micro-behaviors need to be enlarged to support a wider variety of modeling projects.

We chose *NetLogo* as the platform because of its ease of use and expressive power. It has support from a broad and active community of users including teachers and researchers in a variety of sciences. *NetLogo* is well-suited for our primary audience: university students without computer programming experience. We foresee no technical obstacles to the building a variant of the *BehaviourComposer* based upon a different modeling platform.

Initial tests of the *BehaviourComposer* in a pedagogic setting have been encouraging. Two evaluation studies (MBA students and masters of science students in a management research methods module) were recently conducted and the students built and understood relatively complex models (the second chapter of [Epstein and Axtell 1996]) in less than two hours. Another evaluation study of biology students building models of epidemics is scheduled for later this year. We are also collaborating with a doctoral student to explore how useful our approach is for original research in addition to the primarily pedagogic goals of the project. More usage studies would illuminate many of the open questions around this research. Because the *BehaviourComposer* enables non-programmers to construct models, there is the possibility of introducing this kind of model building to younger students.

We have plans to build a new system based upon these ideas. It will be constructed as a web service and is designed to benefit from the kinds of community contributions and support seen in "Web 2.0" services. Furthermore, we plan to explore the idea of implementing the same micro-behavior in different modeling environments.

## 7. ACKNOWLEDGEMENTS

This work was carried out at the Oxford University Computing Services within the Constructing2Learn Project funded by the JISC Designs for Learning Program.

## References

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

[Cousin et al 2005]
Couzin, I.D., Krause, J., Franks, N.R. & Levin, S.A., “Effective leadership and decision making in animal groups on the move”, *Nature* 433, 513-516.

[Deffaunt et al 2002]
Deffaunt, G., Amblard, F., Weisbuch, G. and Faure, T., “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 and Axtell 1996]
Epstein, J. and Axtell, R., *Growing Artificial Societies Social Science From the Bottom Up,* Brookings Institution Press and MIT Press, 1996

[Kahn 2007a]
Kahn, K., “Comparing Multi-Agent Models Composed from Micro-Behaviours”, M2M 2007, Third International Model-to-Model Workshop, Marseille, France, March 2007

[Kahn 2007b]
Kahn, K., Constructing2Learn Project Web Site, http://dfl.cetis.ac.uk/wiki/index.php/Constructing2Learn

[North el at 2006]
North, M.J., Collier, N.T. and Vos, J. R., “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.

[Railsback et al 2006]
Railsback, S. F., Lytinen, S. L. and Jackson, S. K., “Agent-based simulation platforms: review and development recommendations”, *Simulation* 82: 609-623.

[Reynolds 1987]
Reynolds, C. W. “Flocks, Herds, and Schools: A Distributed Behavioral Model”, *Computer Graphics*, 21(4) (SIGGRAPH '87 Conference Proceedings) pages 25-34.

[Scherer and McLean 2002]
Scherer, A. and McLean, A. "Mathematical models of vaccination", *British Medical Bulletin* 2002, 62 187-199

[Simon 1962]
Simon, H. “The Architecture of Complexity”, *Proceedings of the American Philosophical Society*, 106: 467-482

[Wilensky 1999]
Wilensky, U. (1999) *NetLogo*, Center for Connected Learning and Computer-Based Modeling, Northwestern University, http://ccl.northwestern.edu/*NetLogo*/

## 8. AVAILABILITY OF THE *BEHAVIOURCOMPOSER* AND MICRO-BEHAVIORS

A beta version of the *BehaviourComposer* is available for download from http://dfl.cetis.ac.uk/wiki/index.php/Beta_testing.

Micro-behaviors, sample models, and documentation are available at http://dfl.cetis.ac.uk/wiki/index.php/Constructing2Learn.

## 9. BIOGRAPHY

Ken Kahn has been engaged in research in computer programming since before he received his doctorate from MIT in 1979. After exploring programming languages for children he turned towards the design and implementation of very high-level programming languages embodying ideas from object-oriented programming, logic programming, constraint programming, concurrent programming, distributed computing, and visual programming. In 1992, Ken returned to programming languages for children when he founded Animated Programs. He designed and built ToonTalk, an animated programming language for children. He is currently a senior researcher at Oxford University where he is leading the Constructing2Learn Project and is a visiting fellow and researcher at the London Knowledge Lab. He will soon be leading the Modelling4All Project which aims to bring the ideas reported here into a “Web 2.0” setting.