LevelSpace: A NetLogo Extension for Multi-Level Agent-Based Modeling

Arthur Hjorth[a] , **Bryan Head**[b] , **Corey Brady**[c] and **Uri Wilensky**[b]

[a]Department of Computer Science, Aarhus University, Denmark; [b]Northwestern University, United States; [c]Vanderbilt Unviersity, United States

Journal of Artificial Societies and Social Simulation **23** (1) 4
<http://jasss.soc.surrey.ac.uk/23/1/4.html>
DOI: 10.18564/jasss.4130

Received: 14-Mar-2018 Accepted: 22-Nov-2019 Published: 31-Jan-2020

Multi-Level Agent-Based Modeling (ML-ABM) has been receiving increasing attention in recent years. In this paper we present LevelSpace, an extension that allows modelers to easily build ML-ABMs in the popular and widely used NetLogo language. We present the LevelSpace framework and its associated programming primitives. Based on three common use-cases of ML-ABM – coupling of heterogenous models, dynamic adaptation of detail, and cross-level interaction - we show how easy it is to build ML-ABMs with LevelSpace. We argue that it is important to have a unified conceptual language for describing LevelSpace models, and present six dimensions along which models can differ, and discuss how these can be combined into a variety of ML-ABM types in LevelSpace. Finally, we argue that future work should explore the relationships between these six dimensions, and how different configurations of them might be more or less appropriate for particular modeling tasks.

Keywords: Multi-Level, Agent-Based Modeling, Modeling Tools, Netlogo

Introduction

Agent-based models are typically conceptualized and coded as self-contained, separate units, each affording an in-depth 'thinking and analysis space' of *one* particular phenomenon. They are modeled as the interactions between two levels: a micro- (or agent-) and a macro- (aggregate-) level (e.g. Bar-Yam 1997; Epstein & Axtell 1996; Mitchell 2009; Wilensky & Rand 2015). This approach has been useful across many disciplines and domains, but has shown very useful in the social sciences where the micro-level often represents the states and behaviors of social organizations or individual people (e.g. Epstein 1999, 2006; Gilbert & Terna 2000). While the good reasons for restricting models to two levels are manifold, this comes at a cost: by reducing a phenomenon to two levels, we necessarily either exclude processes and entities that do not fit at these levels, or we are forced to abstract them into proto-agents or other low-fidelity simulations.

For instance, in the classic Segregation (Schelling 1969) model, we are restricted to two levels: houses, represented by space, and individuals, who make decisions about whether to move or not. However, having only these two levels of simulated processes cuts out things that lie *outside* of the model, like the impact that segregation might have on neighborhood schools, or things that are *inside* the model but at a more detailed level like the decision process "within" each agent of where to move, and why.

In this paper, we present LevelSpace, a NetLogo extension that provides a simple set of NetLogo primitives for connecting models and building multi-level or multi-model models. We provide examples of how modelers can use LevelSpace to elaborate on processes within existing models, or to connect phenomena, by letting theoretically infinitely large systems of models communicate and interact with each other. Finally, we present six dimensions for describing different kinds of inter-model interactions that LevelSpace facilitates and discuss how these six dimensions help us better describe NetLogo models built with LevelSpace.

Uses of Multi-Level Agent-Based Modeling

Recently, there has been an increasing interest in Multi-Level Agent-Based Modeling, or ML-ABM. Morvan (2012) provides a survey of applications, methods, and approaches for the construction of ML-ABMs[1]. In this survey, Morvan describes three types of problems that ML-ABM languages or packages are typically used to address:

- Coupling of heterogeneous models – Most systems do not exist in isolation and, thus, there is sometimes a need to simulate multiple interacting systems. Furthermore, the forms that the models of the interacting systems take may differ wildly. One example of such a coupling that we will discuss throughout the paper is a 2-model system in which we couple a climate change model with a model showing an ecological system. The timescale for dynamics within an ecological model might be measured in hours or days, whereas in a climate model it could be in years, decades, or centuries. Furthermore, the models must be able to have significantly different topologies, components, and types of interactions. Thus, one needs to be able to couple diverse models such that the needs and ontology of each model are respected.
- (Dynamic) adaptation of level of detail – Computationally modeling a system at fine-grained levels of detail can often lead to performance problems. To overcome this, one may wish to model the system at a high-level, and simulate a low-level view at particular locations or when particular events occur, and only stay there for the duration of those events.
- Cross-level interaction – Systems often consist of multiple levels or scales of interactions. There is sometimes a need for explicit representations of interactions between different scales in a system.

Languages and packages typically take one of three different approaches to addressing these problems. First, some frameworks achieve multi-level capabilities by allowing agents to contain or be defined by lower-level agents. For example, SWARM (Minar et al. 1996) was not only one of the first ABM frameworks, it also has (limited) multi-level capabilities. Agents in SWARM can be defined by swarms of lower-level agents. However, due to the limitations of inter-level interactions in SWARM, Morvan (2012) does not regard it as true a ML-ABM tool. Another example is ML-RULES (Maus et al. 2011), a rule-based, multi-level framework focused on cell biology. It uses a chemical-reaction-equation-like language to describe interactions between "species" and allows species to contain micro solutions of other species. ML-Rules can be used to address cross-level interaction, but not heterogeneous model coupling or dynamic adaptation of detail. Finally, GAMA (Drogoul et al. 2013) is a sophisticated and actively developed ABM framework that has multi-level capabilities. In GAMA, every agent has a defined geometry that forms a space that other agents can reside in. Moreover, information and agents can pass between levels. This allows GAMA to address cross-level interaction and, to some degree, dynamic adaptation of level of detail, but not heterogeneous model coupling.

Second, other frameworks allow multiple models, spaces, or environments, that each contain agents, to be *linked* together. This approach offers significant flexibility (and is the one LevelSpace

adopts). For example, SPARK (Solovyev et al. 2010) is a multi-scale ABM framework designed for biomedical applications. The framework consists of agents, links between agents, spaces which contain the agents, data layers which overlay spaces with quantitative values, and an observer which is responsible for, e.g., scheduling. A model may have multiple spaces of varying scales and topologies. Furthermore, agents and information can be passed between levels. As such, SPARK can be used to address cross-level interaction in the biomedical domain. Scerri et al. (2010) introduce an architecture for designing ABMs in a modular fashion such that each module handles a particular aspect of the system. The modules take the form of ABMs that are synchronized with a "Time Manager", with interactions between the models handled by a "Conflict Resolver" that ensures divergences in the modules are handled gracefully. As such, the architecture is particularly focused on handling heterogeneous model coupling, though it can also handle cross-level interactions that fit within this paradigm. Finally, Morvan et al. (2010) adapts the IRM_4S meta-model (Michel 2007) to handle multiple levels. This adaptation, called IRM_4MLS, describes agents, levels that the agents inhabit, influences that the agents exhibit, and reactions the agents have to influences. Morvan & Kubera (2017) have since refined IRM_4MLS into the new SIMILAR meta-model. Both IRM_4MLS and SIMILAR are targeted at solving cross-level interaction and heterogeneous model coupling problems.

Finally, some frameworks provide multi-level capabilities by imposing a strict, multi-level structure. For instance, one of the earliest architectures, GEAMAS (Marcenac & Giroux 1998) broke models into three different layers. The highest level defines how agents are organized in the model via a "network of acquaintances", as well as the input parameters and output measures of the model. The mid-level includes "high-level, cognitive agents", which act as groups of low-level agents that may be identified as wholes. The third level includes simple, reactive agents. Thus, GEAMAS was targeted specifically at the problem of cross-level interaction rather than the other two problems.

Thus, while many meta-models, frameworks, and architectures have been developed, to date none of them address all three ML-ABM problems. Furthermore, many of them are domain specific and thus are not appropriate for a more general ABM-application. Finally, a number of them employ highly specialized formalisms that severely limit the potential for wide adoption. We have designed LevelSpace not only to address all three ML-ABM problems, but also to preserve the flexibility of NetLogo and maintain its "low-threshold, high-ceiling" philosophy, which enables it to support the thinking and inquiry of a wide range of modelers, from young learners to experienced research scientists.

In the following, we first present and describe LevelSpace. We then demonstrate with three examples how LevelSpace can be used to create models that each address one of Morvan's (2012) ML-ABM problems. To keep these examples simple and avoid relying on specialized domain knowledge, all three examples will extend the relatively well-known and simple Wolf Sheep Predation (WSP) model (Wilensky 1997) which shows population dynamics in a three-tiered ecosystem consisting of predators, prey, and grass.

LevelSpace: A *Language* for Connecting, Expanding, and Exploring .ogo Models

LevelSpace (Hjorth et al. 2015) is an extension for NetLogo (Wilensky 1999) that allows modelers to construct ML-ABMs within the NetLogo language and modeling environment. NetLogo is one of the most widely cited ABM languages in both the natural and social sciences (Hauke et al. 2017; Railsback et al. 2017; Thiele et al. 2012), and its low-threshold, high ceiling design commitment makes it a powerful tool for eliciting complex systems thinking in novice-modeler learners as well

(Tisue & Wilensky 2004). In this capacity, NetLogo has been used for learning in domains as diverse as physics (Sengupta & Wilensky 2009), biology and evolution (Centola et al. 2000; Levy & Wilensky 2005; Stieff & Wilensky 2003; Wagh & Wilensky 2012, 2013, 2014; Wilensky & Reisman 2006), chemistry (Levy & Wilensky 2005; Stieff & Wilensky 2003), material sciences (Blikstein & Wilensky 2008, 2010), and robotics (Berland & Wilensky 2005, 2006; Blikstein & Wilensky 2007), introductory computer science (Dickerson 2015; Gilbert 2008; Stonedahl et al. 2009) and increasingly in social sciences education as well (Guo & Wilensky 2018; Hjorth et al. 2018; Hjorth & Wilensky 2014, 2018).

LevelSpace extends the capabilities of NetLogo to enable ML-ABM across modeling and learning in all these domains. It allows modelers to dynamically and programmatically *open, run, control, and close* an arbitrary number of NetLogo models during runtime from *inside* an arbitrary number of other NetLogo models, using the NetLogo language itself to do so. LevelSpace allows modelers to think of models as *agents* by letting them manipulate models as they would any other type of agent in NetLogo. In other words, models are created inside other models in the same way as agents are created inside models. In the rest of the paper we will refer to models that have been created by another model as respectively 'child model' and 'parent model'. A child model can be any NetLogo model, even one that uses LevelSpace. Thus, child models that use LevelSpace can open their own child models, allowing for an arbitrary number of children, grand-children, great-grand-children, etc. to the original "top-level" model.

LevelSpace supports two different kinds of child models: *Interactive* Models, and *Lightweight* Models. The former will present the user with its entire Graphical User Interface (GUI), including all interface 'widgets' (plots, buttons, choosers, etc.). The latter, by default, runs in the background with no user GUI; but even here, it is possible to show the view-widget if a modeler is interested in seeing what goes on behind the scenes. However, *Lightweight* Models will not allow the user to see or manipulate interface widgets, inspect agents, etc. Each of these two child model types is useful in different contexts. Each of our three examples of LevelSpace ML-ABMs use child models in a distinctive way, based on the design and modeling needs of that particular example, and to illustrate the flexibility and expressiveness of LevelSpace.

Design considerations about representations of models in LevelSpace

At the core of LevelSpace are representations of models *inside other models* that can manipulate or retrieve data from each other. At the core of our *design decisions*, therefore, was the question of how to represent models and collections of models.

Early in the design process, we considered the various options for representations of models and collections of models. One such option, maybe the most obvious one, was to create a Model-object and a Modelset-object that would mirror the Agent-object and the Agentset objects in NetLogo's Java and Scala code. We began implementing this approach on one branch of our development work to explore whether this would work. We quickly saw that it had certain advantages: for instance, storing all models as objects inside the extension allowed us to always ensure that "dead" models were removed on-the-fly from Modelsets, similarly to how NetLogo removes dead turtles. Additionally, having models as objects also allowed us to provide more information about them in the Command Center, similarly how to NetLogo can provide information about a turtle if you "report" it. However, there was a serious disadvantage to this approach: NetLogo does not currently allow its built-in primitives to interact with objects that are defined in extensions. This meant that we would have to reimplement every single primitive that would interact with Model or with Modelset objects. More importantly, it meant that modelers and learners would have even more primitives to learn, and would need to learn when to use the extension specific primitives, and when to use the built-in ones.

We decided to make *minimizing the number of new, necessary primitives* without sacrificing modeling power our primary design objective. For this reason, we decided to design LevelSpace to use numbers to represent models (henceforth called "IDs" or "model IDs") and regular NetLogo-lists to represent collections of models. These model representations, as the reader will see in the following description of the extension primitives, are designed to emulate the traditional representation of turtles and other NetLogo agents as closely as possible while being able to utilize NetLogo's existing primitives for iterating over, sorting, filtering, and manipulating lists.

LevelSpace Primitives

As LevelSpace is built on NetLogo's Extension API and Controlling API, it deploys only a few powerful primitives used for keeping track of models, and for running commands and reporters in them. As mentioned, a core design principle of LevelSpace is to preserve as much of the existing NetLogo syntax and semantics as possible, both in the core NetLogo, and in other extensions like the xw-extension[2]. These primitives therefore bear a strong resemblance to existing NetLogo primitives that serve analogous modeling functions for simple agents.

In this section we will list and describe the most commonly used primitives that are part of the LevelSpace extension. For an exhaustive list of primitives, please see the GitHub wiki page[3]. We will give a brief example or two, discuss how and when to use each primitive, and discuss advantages and disadvantages of the design of the primitive where relevant.

ls:models

In order to keep track of child models, the IDs of all child models contained within a model can be reported with the `ls:models` primitive. It returns a list of all IDs in the order of time of creation, and LevelSpace keeps track of models and removes discarded ones, so that `ls:models` always returns only currently open models. As mentioned, using lists helps integrate the execution and processing of LevelSpace code, because it allows easy integration with existing list related primitives, like `map`, `reduce`, `sentence`, and `foreach`. Any model's child models can only be retrieved using `ls:models` from within the model itself. In other words, if Model A has a collection of child models, and Model B has a collection of child models, then the returned value of `ls:models` will depend on in which model you run this reporter. This is similar to how the returned value of reporting any agent value in NetLogo will depend on which agent you report it from. This also means that if a modeler wishes to retrieve grandchild models, they would need to run `ls:models` in the context of their child model. (We will show how to report values below.)

ls:ask

Like the rest of the NetLogo language, LevelSpace is polite. So, we *ask* child models to do things. `ls:ask` takes a model ID (a number), or a list of model IDs, and a block of code. So

```
ls:ask 0 [ create-turtles 100 ]
```

would ask model 0 to create 100 turtles, while

```
ls:ask [0 1 2] [ create-turtles 100 ]
```

would ask the models with IDs 0, 1 and 2 to do so. We can also pass arguments into `ls:ask` using NetLogo's standard lambda-syntax. This helps us dynamically change the values that we base the

LevelSpace model interactions on, e.g.

```
let number-of-turtles 50
let turtle-colors (list red green blue)
(ls:ask 0 [ [ n c ] -> create-turtles n [ set color one-of c ] ] nu
```

would ask model 0 to create 50 turtles and then set each turtle's color to one of those provided in `turtle-colors`. Importantly. `ls:ask` only asks models that are directly available to a model, i.e. its own child models. Consequently, if a model wishes to ask a grandchild-model to do something, we would need to explicitly pass the ask request "down the family tree", like this:

```
ls:ask 0  [
;; within the context of this child model, 0 refers to its own chil
  ls:ask 0  [
    do-something
  ]
]
```

ls:of / ls:report

There are two ways of reporting data from models. The first, `ls:of`, uses 'infix' syntax but does not take arguments. The second, `ls:report`, uses 'prefix' syntax but allows for use of arguments. Just like `ls:ask`, `ls:of` and `ls:report` take either a single model ID, or a list of model IDs. So,

```
[ count turtles ] ls:of 0
```

will report the number of turtles in model 0. This primitive was designed to work like NetLogo's built-in primitive `of` and uses the same syntax. Similarly, if we wanted to report the number of turtles in three models, 0, 1, and 2, we could do:

```
[ count turtles ] ls:of [ 0 1 2 ]
```

This would return a *list* of numbers, representing the count of turtles in the models with IDs 0, 1, and 2, respectively. The reported list will be ordered and aligned with the input list.

Another way of reporting values is to use the `ls:report` primitive. This works analogously to NetLogo's built-in primitive for running strings as code, called run-result, and allows for the use of arguments:

```
(ls:report 0 [ a-color -> count turtles with [ color = a-color ] ] blue)
```

Note that because arguments are optional, it is necessary to enclose uses of `ls:report` in parentheses when providing arguments.

When passing in a list of model IDs, `ls:of` and `ls:report` return results in the order of the list so that you can have many models run a reporter in parallel and still be able to match up models and their respective result. This could feel like a departure from the regular of-primitive in NetLogo where

```
[ xcor ] of turtles
```

will return the x-coordinates of all turtles in a random order each time it is evaluated.

However, `of` actually returns values in the same order as it is given turtles. What causes the randomization of order is the fact that `turtles` is an agentset, which will always return agents in a random order. In other words, the difference between the behavior of `of` and `ls:of` and `ls:report` is due not to a difference in the implementation of the LevelSpace-primitives, but to the fact that we store model IDs in lists, which are ordered.

`ls:of` and `ls:report` can be used to retrieve data from grandchildren by embedding them inside themselves. Both put a reporter in the context of a particular child model, just like regular of puts a reporter in the context of an agent or agentset. This means that any reporter will be evaluated for that child model.

```
;; this reports 'count turtles' from the top-level model's child nc
[ [ count turtles ] ls:of 4 ] ls:of 0
```

As in regular NetLogo code, the reporters are evaluated beginning with the inner blocks, so the code above will report the number of turtles in model 0's child model with ID 4.

ls:let

`ls:let` allows the parent model to store information in a variable that may then be accessed by a child model. It is similar to the let primitive in NetLogo. For example:

```
ls:let number-of-turtles 100
ls:ask 0 [ create-turtles number-of-turtles ]
```

will first assign 100 to the LevelSpace temporary variable called `number-of-turtles`, and then pass that to its child model. Consequently, the code above will create 100 turtles in the child model with ID 0.

`ls:let` variables can be used exactly as any other local variable in the child model's context. However, the parent model may not change (or even read) the value of a `ls:let` variable once it is set. This is intentional:
`ls:let` variables should be used only to pass information from the parent to child, and not used for any computation in the parent. With `ls:let`, sharing information with child models becomes just as natural as using local variables to pass information from one agent to another.

As a stylistic side note, `ls:let` can replace all uses of arguments, and often results in slightly more human readable code, whereas using arguments can be less verbose.

Importantly, `ls:let` exists in the context of a parent. This means that unlike a normal NetLogo let-variable which exists inside the scope of a full block, the value of a `ls:let` cannot be passed down through grand children without having to use `ls:let` again. I.e.

```
ls:let num-turtles count turtles
ls:ask 0 [
  ls:ask 0 [
  create-turtles num-turtles ;; WILL ERROR
  ]
]
```

will fail because `num-turtles` only exists in the top-level model. Rather, the top-level model would need to ask its child model to reassign the `ls:let` in its own context, and then pass that into its own child model – the grandchild of the original model:

```
ls:let num-turtles count turtles
ls:ask 0 [
  ;; bind the value to a ls:let in the context of the child model
  ls:let child-num-turtles num-turtles
  ls:ask 0 [
    create-turtles child-num-turtles ;; this works
  ]
]
```

ls:with

Often, we want to filter models, similarly to how we filter agents in NetLogo with the built-in `with`-primitive. For instance, we may want to ask only those of our models that satisfy a particular condition – e.g. only those with more wolves than sheep – to do something. For example, suppose we wanted to call the GO-procedure in only models where wolves outnumber sheep. The following code combines `ls:ask` and `ls:models` with `ls:with` to achieve this:

```
ls:ask ls:models ls:with [ count wolves > count sheep ] [ go ]
```

ls:create-interactive-models / ls:create-models

These primitives open child models. Recall from above that LevelSpace allows for two different kinds of child model: *interactive* models that present a full interface, and *lightweight* (or 'headless') models that, by default, run in the background (but that can present a model view as well, if needed). Both primitives take as an argument the number of models to open and a path to the .nlogo file that the modeler wishes to open – this allows it to work analogously with NetLogo's built-in `user-file` primitive. Additionally, both primitives can also take an anonymous command as an argument. When used, this command receives the newly created model's ID as an input. Importantly, this command block is run in the context of the parent model, and not in the child model. This differs from NetLogo's regular `create-*` primitives, because the optional command block here is run in the context of the agent that was just created. However, we made this change to make it easy for the modeler to store a reference to the newly created model in a variable. So, assuming the existence of two global variables

```
globals [ wolf-sheep-predation climate-change ]
```

the following code:

```
ls:create-models 1 "Wolf Sheep Predation.nlogo" [ model-id ->
  set wolf-sheep-predation-model model-id
]
ls:create-models 1 "Climate Change.nlogo" [ model-id ->
  set climate-change-model model-id
]
```

loads an instance of each of the two models and assigns them to their respective variables. This not only helps keep track of models but can also be used as a device for writing more easily readable code using illuminating variable names for child models instead of model ID numbers.

ls:close

This primitive simply takes a number or a list of numbers and closes the models associated with those IDs. If the models that are closed contain LevelSpace and child-models of their own, these will also be closed recursively. All associated objects will be garbage collected by the JVM when needed. Consequently, this primitive is imperative for memory management. As mentioned, `ls:close` automatically removes a child model from its parent's `ls:models`.

ls:show/hide

These primitives also take a number or list of numbers and show or hide the view of the corresponding models. For lightweight/headless child models, this window contains only the view widget, while for interactive/GUI child models it contains the entire interface. When hidden, the child models will keep running in the background. This saves all drawing calls in NetLogo, and automatically sets each models' speed slider to maximum. Consequently, `ls:hide` can be used to make a LevelSpace model run much faster in those cases where viewing the model during runtime is not necessary.

ls:reset

This primitive clears LevelSpace, closing down all existing child models (and, if necessary, any descendant models), and readies the LevelSpace extension for a new model run. This also resets the "serial number" of child models, so the next child model created by the model will have ID 0. In our experience, this command will typically be used in the setup procedure of the top-level parent model.

ls:name-of / ls:path-of

When dealing with many different types of models, it is often useful to be able to identify a model on-the-fly. `ls:name-of` and `ls:path-of` return, respectively, the name of the .nlogo file, and the full path of the .nlogo file that was used when loading a particular model. So that

```
ls:create-models 1 "Wolf Sheep Predation.nlogo"
show ls:name-of 0
```

will show “Wolf Sheep Predation.nlogo” in the command center. This primitive illustrates an example of when it can be useful to combine NetLogo’s built-in list primitives with LevelSpace primitive: Imagine that a model has loaded some number of child models, including some Wolf Sheep Predation-models, and that we for some reason only want to call GO in the Wolf Sheep Predation child models. This can be achieved by combining `ls:ask`, `ls:models`, and `ls:name-of` with NetLogo’s built-in `filter` like this:

```
ls:ask filter [ id -> ls:name-of id = "Wolf Sheep Predation"] ls:mc
```

ls:uses-level-space?

Calling an `ls:*` primitive in a model that does not have the LevelSpace extension loaded will result in a runtime error. It is therefore often useful to be able to check whether a child model has the extension loaded. This primitive takes a model ID and returns true if the child model is running LevelSpace. This, too, can be combined with `filter` to, e.g. only pass a LevelSpace-specific command or reporter to those child models that are able to run it.

```
;; only report count turtles from child models with LevelSpace
[ count turtles ] ls:of filter [ id -> ls:uses-level-space? id ] ls
```

Managing Order Across Levels in LevelSpace

The multi-model, multi-level nature of LevelSpace means that there are some differences between the way in which code is executed, and the way in which we ensure synchronicity between models in LevelSpace compared to normal NetLogo. Understanding how to ensure that code is executed in the desired order is important in modeling, and we therefore dedicate a section on it here.

LevelSpace’s child models can, as mentioned, also load LevelSpace extension, and create their own child models. As mentioned in the previous section, in order for a grandparent model to ask a grandchild model to do something, we can embed an `ls:ask` inside the code block that we pass down the family line with `ls:ask`, like this:

```
ls:ask 0 [ ls:ask grandchild-model [ do-something ] ]
```

An important difference between LevelSpace and regular NetLogo is that the numbers reported from `ls:models` do not refer to the *child model object*, but simply to the index of that child model in its own parent’s list of child models. It would therefore not be possible to report the number up to a grandparent and run code directly from the grandparent. i.e. this code

```
;; reports a list of lists of child id numbers relative to the top-
let grand-children [ ls:models ] ls:of ls:models
ls:ask grand-children [ do-something ]    ;; DOES NOT WORK
```

will not work because the numbers that are stored in grand-children refer to the model IDs relative to their parent, and not relative to the grandparent. One would therefore need to ask each child to ask its children to `do-something` (as we will show later in par. 5.7).

Concurrency in LevelSpace

LevelSpace extends NetLogo's approach to scheduling behavior by viewing models as agents. In NetLogo, if a modeler wants a set of agents to walk forward and then show how many neighboring turtles they have, the modeler will write

```
ask turtles [ forward 1 ]
ask turtles [ show count turtles-on neighbors ]
```

In this case, all turtles would first move forward, and *then* show how many neighbors they have. No turtle would report their neighbor count until they have all finished moving. However, if we write:

```
ask turtles [
  forward 1
  show count turtles-on neighbors
]
```

each turtle would move forward and immediately show how many neighboring turtles they have. This would create a situation in which turtles count each other as neighbors while being out of sync. This is not possible in LevelSpace because a query about other models would need to go through the parent, and the child is not able to ask the parent to run a reporter while it itself is running code. Instead, if the modeler needs child models to react to each other's information or states, the modeler would need to break down the code into chunks, i.e.

```
;; change the state of child models
ls:ask ls:models [ do-something ]
;; retrieve the information and bind it to `foo`
ls:let foo [ bar ] ls:of ls:models
;; pass the information into child models to act upon
ls:ask ls:models [ do-something-with foo ]
```

While this may be an edge case, it is an example of something that is different between single-model NetLogo and LevelSpace and deserves mention.

Just like with a regular ask, LevelSpace executes all commands within a given code block for each model before moving on to the next model. This includes embedded `ls:ask` to child-models. So

```
;; ask all child models, one a time,
ls:ask ls:models [
  ;; to ask THEIR child models, one at a time,
  ls:ask ls:models [
  ;; to do something
  do-something
  ]
]
```

This means that the user has great control, but also full responsibility, for scheduling commands correctly. This requires thinking carefully about how to cascade code down through the family tree of models when using `ls:ask` or `ls:of` to manipulate or retrieve information from grandchild-models.

Parallelism

LevelSpace's `ls:ask` and `ls:of` are designed to run in parallel and will automatically create as many threads as it can in order to execute the code as quickly as possible *for each call*.

```
ls:create-models 100 "amodel.nlogo"
ls:ask ls:models [ create-turtles 1000 ]
;; the following code will not run until all models are done
show [ count turtles ] ls:of ls:models
```

When working with embedded `ls:ask` requests like our example in paragraph 5.7,

```
;; We first ask all of our children
ls:ask ls:models  [
;; to ask all their children
  ls:ask model-id2  [
    do-something
  ]
;; any code here will only execute once all grandchildren of the tc
```

each child will, in parallel, *ask their* children, in parallel, to `do-something`. Only when all the grandchildren are done will the following code be executed.

For most uses, this will not make any difference to the user, other than running LevelSpace code faster.

Non-Language Features of LevelSpace

LevelSpace introduces some new features to NetLogo's IDE that do not relate specifically to the design of primitives. Further, it introduces a new data type called the 'code block'. These new

features were designed to make it easier to deal with the particular challenges introduced by ML-ABM.

LevelSpace as a development platform

Developing a LevelSpace model breaks with a fundamental assumption in the NetLogo IDE – that there is always one, and only one model. When building LevelSpace models, one often works in several different .nlogo files at a time. We have therefore made it easy to load, view and/or edit more models at the same time in NetLogo. We did this by designing a few changes to the NetLogo IDE that happen seamlessly as soon as a modeler imports the LevelSpace extension. The first of these changes is that a "LevelSpace" drop-down menu is added (Figure 1).

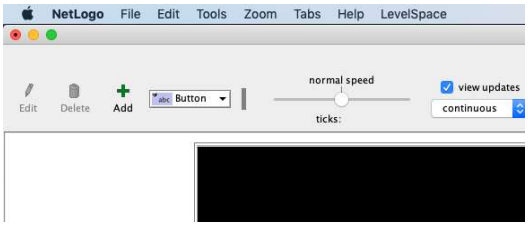

Figure 1. As soon as LevelSpace is loaded, the LevelSpace menu appears.

The menu (Figure 2) has three items. The first one lets the modeler open the code of an existing NetLogo model. This is often useful, either to make edits to a child model, or to be able to easily access the model to look up the names of procedures, variables, etc. The second item contains a list of all models that have been opened by LevelSpace since it was first loaded by using the èxtensions` keyword. The third allows a user to create a blank model which is often useful if a modeler is building a new LevelSpace model from scratch.

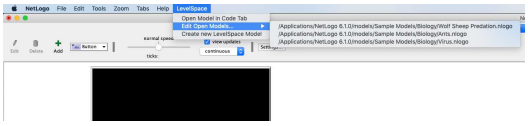

Figure 2. The LevelSpace menu, expanded.

Any of these options will open a new tab that includes the code for this model (Figure 3). However, it is not possible to make changes to the Interface of child-models. If a modeler needs to make changes to the interface, they would need to open another NetLogo application instance.

Figure 3. Example of many models open in the same NetLogo window.

Namespacing between models and the code block

Of course, when writing NetLogo code for different models there are immediate namespacing issues. To address these, LevelSpace introduces a new datatype to NetLogo called code block. NetLogo code is interpreted as a code block if it is surrounded by hard brackets, and if it is passed into any of the `ls:-` primitives that accept it. The primary difference between the code block and the usual command block or reporter block is that it is not evaluated during compile time. This makes it possible to refer to variable names in the child model in the code belonging to the parent model without getting compilation errors. In the following, assume that the child model has a global variable named `foo`.

```
;; foo does not exist in the parent model so the line below will nc
set foo "bar"
; foo does not exist in the parent model, but this will compile bec
ls:ask ls:models [ set foo "bar" ]
```

It also makes it possible to disambiguate namespace collisions between parent and child. Unlike in single-model NetLogo, where a turtle variable and a global or link variable cannot share names, it is possible in LevelSpace to have namespace collisions between a parent model and its child models – in fact it is impossible not to have namespace collisions because of built-in variables and reporters. For instance, all models in NetLogo have a reporter called `turtles`, but the code block makes it clear which model's turtles we refer to:

```
show turtles                      ; this is the parent's turtles
show [ turtles ] ls:of a-model    ; this is the child's turtles
```

However, because the code block is not evaluated until runtime, this also means that it is possible to refer to variable names that do not exist in the child model, without getting a compilation error in the NetLogo IDE as one would if coding with a single NetLogo model. As a consequence, if *foo* does not exist in the child model, LevelSpace throws an ExtensionError but only during *runtime* (Figure 4).

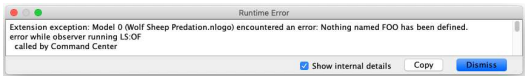

Figure 4. A runtime Extension exception that would usually occur at compile time with regular NetLogo.

Examples of Multi-Level Agent-Based Models with LevelSpace

As described above, Morvan's (2012) survey shows that ML-ABM has been used to solve three general kinds of problems. In the sections that follow, we will provide examples addressing each of these problem types by using LevelSpace to construct multi-level ABMs that extend the Wolf-Sheep Predation NetLogo Model (Wilensky 1997).

LevelSpace Example 1: Coupling of heterogeneous models – Population dynamics and climate change

In our first example, which we call Population Dynamics and Climate Change[4], we expand on the Wolf Sheep Predation (WSP) model by linking it in a ML-ABM system with a model of Climate Change (Tinker & Wilensky 2007). The Climate Change (CC) model shows how clouds and the presence of greenhouse gases in the atmosphere contribute to the containment of energy from the sun and how this affects the global temperature. Each of the two models have a clear purpose in their own right: the WSP model allows modelers to think about the oscillation of population sizes in ecosystems consisting of sheep and wolves, and the CC model helps learners understand how the mechanics of photons, greenhouse gases and infrared radiation contribute to global warming. But consider this non-exhaustive list of ways in which these systems affect each other:

- The temperature in the CC model affects how quickly grass grows back in the WSP model.
- The grass in the WSP model absorbs CO2 from the atmosphere.
- Animals in WSP – both wolves and sheep – contribute greenhouse gases in the CC model via expiration and flatulence when they metabolize food.
- The proportion of grass to proportion of dirt patches in WSP affects the albedo[5] in the CC model, which in turn affects how much visible light from the sun is absorbed by Earth and how much is reflected.

Setting up these relationships is easy with LevelSpace. The following commented code for our new parent model shows how we import the LevelSpace extension (ls), define global variables `wolf-sheep-predation` and `climate-change`, for the IDs of each of the two child models, and create a SETUP-procedure for the parent model. Note that no changes are required to *either* of the WSP or CC models for them to interact as child models in this new ML-ABM:

```
extensions [ ls ] ;; load the LevelSpace extension
;; define a variable for each of our models
globals [ climate-change wolf-sheep-predation ]

to setup
  ls:reset  ;; close child models from any previous runs
  clear-all
  ;; open CC and assign it to climate-change
  (ls:create-interactive-models 1 "Climate Change.nlogo" [ new-mode
    set climate-change new-model ])
    ;; open WSP and assign it to wolf-sheep-predation
  (ls:create-interactive-models 1 "Wolf Sheep Predation.nlogo" [ ne
    set wolf-sheep-predation new-model ])
  ls:ask climate-change [
    no-display
    setup
    repeat 20 [ add-co2 ]
    repeat 3 [ add-cloud ]
    ;; the CC model requires a bit more initialization, as it doesn
    no-display
    repeat 10000 [ go ]
    display
  ]
;; turn on 'grass' in the WSP model, and run setup
  ls:ask wolf-sheep-predation [ set model-version "sheep-wolves-gra
end
```

In order to program these interactions between the two systems, we can write a GO-procedure in the parent model that uses LevelSpace's primitives. We

- make the global variable `grass-regrowth-time` in the WSP model vary as a function of the `temperature` global variable of the CC model,
- ask the wolves and sheep with `energy` greater than 25 in the WSP model to call the `add-co2` procedure in the CC model (interpreting 'co2' in the CC model now to represent all

greenhouse gases, including methane),
- ask the grass in WSP to call the `remove-co2` procedure in CC, and finally
- dynamically change the `albedo` global variable in the CC model as a function of the proportion of patches in WSP that have grass.

Using LevelSpace, this could be written like this:

```
to go
  ls:ask ls:models [ go ]
  ;; calculate and set new temperature
  ls:let new-regrowth-time 25 + ( abs [ temperature - 55 ] ls:of cl
  ls:ask wolf-sheep-predation [
    set grass-regrowth-time round new-regrowth-time
  ]
  ;; count gassy animals and have them add greenhouse gases
  ls:let count-gassy-animals [ count turtles with [ energy > 25 ] ]
  ls:ask climate-change [ repeat count-gassy-animals [ add-co2 ] ]
  ;; grass binds co2
  ls:let amount-of-grass [ count grass ] ls:of wolf-sheep-predation
  ls:ask climate-change [ repeat amount-of-grass / 25 [ remove-co2
  ;; find proportions of grass and dirt and set albedo
  ls:let new-albedo  [ count grass / count patches ] ls:of wolf-she
  ls:ask climate-change [ set albedo new-albedo ]
end
```

This syntax should look familiar to NetLogo users and modelers, and it should also be legible to agent-based modelers who are not deeply familiar with NetLogo.

As mentioned, WSP and CC in their original form serve particular purposes, and they support inquiry into different questions on their own than when they are linked to form a ML-ABM. As such, these models can address questions about how ecosystems and climate change mutually affect each other, and, in the tradition of butterfly effects, how individual sources or sinks of CO_2 might ultimately affect the life of entire ecosystems.

Importantly, when we begin to connect individual agent-based models in a multi-model model, they raise new questions and support us in exploring new collections of emergent phenomena. From a modeling-as-methodology perspective, this is important because it potentially allows us to validate and verify individual models before connecting them to a larger model-system, thus allowing us to add both breadth and depth in our modeling endeavors.

Second, it does so without the disadvantage of rigidly bounding the modeled phenomena: if we (or anyone else) should want to expand on our ML-ABM by adding new models or by changing the ways in which the models interact, LevelSpace makes this easy. We believe these are important improvements to modeling both as a scientific practice and as a reflective process. LevelSpace thus broadens the scope of the possible conversation that the scientific community can have around a model (or model system), and it allows modelers to easily expand on the otherwise more rigid boundaries that models draw around a particular phenomenon.

LevelSpace Example 2: (Dynamic) adaptation of the level of detail ¬– Wolf Chases Sheep

When a wolf and sheep meet in the Wolf Sheep Predation model, the wolf simply eats the sheep. Of course, this is not a realistic portrayal of the predation process. For some purposes we might want to model the chase between a wolf and a sheep every time an encounter like this takes place. However, doing so within the original WSP model would put the temporal and the spatial scales of the model at odds with themselves: the chase between two individuals takes place at a finer granularity of both time and space than the rest of the model.

With LevelSpace, one possible approach is to build a new model that explores the chase at an appropriate temporal and spatial scale, and to invoke this model as a child model whenever a wolf-sheep interaction occurs in the WSP model. To illustrate, we wrote a simple model that we call the Wolf-Chases-Sheep (WCS) model[6] (Figure 5). Whenever a wolf and a sheep meet each other, the Wolf-Chase-Sheep model is initialized with a wolf and a sheep that have the respective headings and relative positions of the wolf and the sheep in the WSP model. The Wolf-Chases-Sheep model simulates the chase process and runs until either the sheep has managed to escape by reaching the edge of the WCS-model's space, or the wolf has caught the sheep. Using LevelSpace, we could resolve whether the wolf catches the sheep, or the sheep escapes in the child model like this:

```
ask wolves [
  if any? sheep-here [
    ;; wolves find potential prey
    let prey one-of sheep-here
    ;; open the chase model
    ls:create-interactive-models 1 "Chase Model.nlogo"[7]
    ;; get the latest model ID and assign it to named variable for
    set chase-model last ls:models
    ;; save the sheep's heading
    ls:let s-heading [ heading ] of prey
    ;; save the wolf's heading
    ls:let w-heading heading
    ;; set the heading of the wolf and the sheep in the chase model
    ls:ask chase-model [
      set wolf-heading w-heading
      set sheep-heading s-heading
      setup
    ]
    ;; run the model until the sheep has escaped or is caught
    while [ not [ done? ] ls:of chase-model ] [
      ls:ask chase-model [ go ]
    ]
    ;; if the wolf caught it, add energy and kill the sheep. Else d
    if [ wolf-caught-sheep? ] ls:of chase-model [
      set energy energy + wolf-gain-from-food
      ask prey [ die ]
    ]
    ls:close chase-model ; close down the model
```

```
    ]
]
```

By simulating the chase process in its own model, we are able to explore some parameters of that process that would otherwise be either difficult or impossible to explore because of the way in which the WSP model imposes particular levels of scale. By 'zooming in' on the process in a separate model with an appropriate set of scales, we are able to add to the original WSP model a representation of the chase process at a finer granularity. This representation is, like any model, a contestable site of inquiry, and it opens a context for exploring how variables like speed or the agility of wolves and sheep might affect their success in predation. This allows us to better understand how these variables affect survival rates at the level of the individual meeting between a wolf and a sheep, *and* how this might affect the population dynamics in the whole ecosystem. For instance, further elaborations on this model could include the energy cost of the chase to both wolf and sheep, and could lead to a more sophisticated cognitive model of wolves deciding whether or not it is worth chasing a particular sheep. This expansion of the model would also better align with more recent developments in our scientific understanding of the role of behavior and decisions in ecosystems.

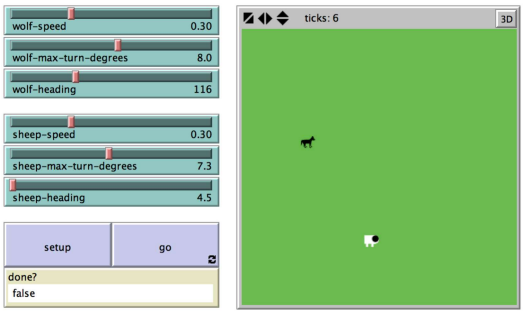

Figure 5. A model in which a wolf tries to catch a sheep.

LevelSpace Example 3: Cross-level interaction – Models as agent cognition

LevelSpace can also be used to supply wolves and sheep with a cognitive model that simulates how these agents reason and behave. Here, we have created a neural network model to fill this role (code available on GitHub[8]). In WSP and using LevelSpace, we ask each wolf and sheep to open its own instance of a neural network model, giving each animal the capacity for unique behavior (Figure 6). At each tick, the animals in WSP send information about their surroundings to their neural network model. The neural network processes the information and sends back a response, which the animal then processes in order to decide how to act. One way to think of this neural network, then, is as a way of predicting what the best course of action is for a given agent. When an animal reproduces, it passes on a copy of its neural network to its child, with the weights between the nodes mutated slightly. Then, because the viability of patterns of responses affects survival, the animals can evolve the behaviors that provide the best chances of survival over generations.

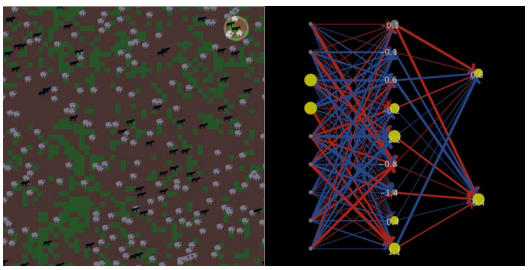

Figure 6. Left: WSP in the middle of a run with a sheep highlighted. Right: The highlighted sheep's neural network from the same moment. Red links are excitatory, blue inhibitory. Thickness indicates weight. Opaque links have been activated by current input. The nodes on the left are the 9 inputs corresponding to the presence of each agent type in each direction. The nodes on the right are the three outputs corresponding to turning left, going straight, or turning right.

Specifically, animals sense nine conditions: the presence of any of three stimuli (grass, sheep, or wolves) in any of three directions (to the left, to the right, or ahead of them). Processing stimuli, the animal's neural network outputs probabilities of predictions of each decision to turn left, turn right, or not turn will be the best decision. Finally, as in the original model, the animal always moves forward and eats anything edible that it encounters.

To set this up, we first define what information each animal will send to its neural network, and how it will deal with the output it gets back. We do this by defining a list of anonymous reporters in our setup procedure that will be used as input for the neural networks as well as the behavior to perform if activated as a list of anonymous commands. Each reporter in the input list detects the presence of a certain agent type in a certain direction from the agent. This could be done with a series of Booleans instead, but this example shows how to combine map and anonymous reporters with LevelSpace primitives to write succinct – though also more advanced, and potentially less readable – code.

```
set inputs (list
  ;; Is there any grass to my left? (BINARY turns the boolean from
  [ -> binary any? (in-vision-at patches (- fov / 3)) with [ pcolor
  ;; Is there any grass in front of me?
  [ -> binary any? (in-vision-at patches 0) with [ pcolor = green ]
  ;; Is there any grass to my right?
  [ -> binary any? (in-vision-at patches (fov / 3)) with [ pcolor =
  ;; Are there any sheep to my left?
  [ -> binary any? other (in-vision-at sheep (- fov / 3)) ]
  ;; Are there any sheep in front of me?
  [ -> binary any? other (in-vision-at sheep 0) ]
  ;; Are there any sheep to my right?
  [ -> binary any? other (in-vision-at sheep (fov / 3)) ]
  ;; Are there any wolves to my left?
  [ -> binary any? other (in-vision-at wolves (- fov / 3)) ]
  ;; Are there any wolves in front of me?
  [ -> binary any? other (in-vision-at wolves 0) ]
  ;; Are there any wolves to my right?
  [ -> binary any? other (in-vision-at wolves (fov / 3)) ]
)

set outputs (list
  [ -> lt 30 ] ;; Turn left
  [ -> ]       ;; Don't turn
  [ -> rt 30 ] ;; Turn right
)
```

Next, we define a reporter that captures the information defined by the inputs list, sends it to the neural network with LevelSpace, and reports the results. The `apply-reals` reporter in the neural network model sets the inputs of the neural network to a list of numbers, propagates those values through the network, and reports the resulting output as a list of probabilities. `brain` is a turtles-own variable that contains a reference to that agent's neural network model ID:

```
to-report sense
  report apply-brain (map runresult inputs)
end

to-report apply-brain [ in ]
  ls:let inputs in
  report [ apply-reals inputs ] ls:of brain
end
```

Finally, we tell the animals to act on the results. Actions are chosen randomly based on the probabilities returned by the neural network:

```
to go-brain
  let r random-float 1
  let i -1
  ;; Get the probability outputs from the agent's brain model-id
  let probs sense
  ;; Randomly select an action based on those probabilities:
  while [ r > 0 and i < length probs ] [
    set i i + 1
    set r r - item i probs
  ]
  run item i outputs ; Run the corresponding action
  fd 1
end
```

Thus, LevelSpace allows modelers to implement sophisticated cognitive models for their agents as independent, modular, agent-based models. This example demonstrates how one can encode the perception of an agent, send that information to the agent's cognitive model, and use the result to guide the behavior of the agent. Furthermore, it allows high levels of heterogeneity among agents, since each agent can be given its own, unique cognitive model. As seen in the neural network example, LevelSpace allows researchers to construct agent-based versions of AI techniques to drive the decision making of their agents. However, LevelSpace also allows for the creation of novel methods of implementing agent cognition. We provide here a briefer example without code (though it is available on GitHub[9]), in which each of the wolves and sheep use a child model to run short, simplified simulations of their local environment to make decisions (Figure 7).

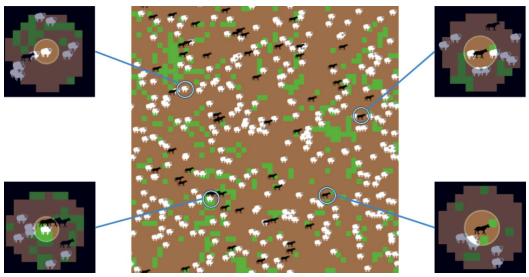

Figure 6. A version of Wolf-Sheep Predation in which agents use explicit agent-based models of their local environment to make decisions. Image from Head & Wilensky (2018)

This example works as follows: each agent has a vision variable that determines its ability to sense its local environment. Each tick, every animal sends its knowledge of the state of its local environment to its own cognitive model. This includes the locations of nearby grass, the locations and headings of the surrounding wolves and sheep, and its own energy level. The animal then runs several short micro-simulations in which agents perform random actions (as in the original Wolf-Sheep Predation), keeping track of how well it does in each run (based on change in energy, and whether the animal died). The animal then selects the action that led to the best outcomes, on average. The rules of the cognitive model are similar to the rules of the original WSP, but have been modified to remove factors that the animals would not know about or would not base decisions off of. For instance, they do not know the energy levels of the other agents, nor do they know about grass regrowth. Thus, these factors are not included in the cognitive model. For full details, please see Head & Wilensky (2018).

This method grants agents powerful and flexible decision making in a natural way. For instance, this method will generally result in a sheep reliably moving to the closest patch of grass. However, if there is a wolf on the closest patch of grass that will likely eat the sheep, this method may result in the sheep running away instead. Then again, if the sheep is about to starve to death unless they eat immediately, they may decide that it is worth risking being eaten by the wolf in order to get the grass. This sort of decision making emerges from the simple method of agents running micro-simulations to evaluate the consequences of actions, based on the locally available information, and the heterogenous states of other agents. Furthermore, in addition to allowing researchers to limit what agents know, the method also gives the researchers two "intelligence" parameters that they may use to control exactly how well their agents behave: the number of micro-simulations, and the duration of each micro-simulation, i.e. how many *ticks* into the future each agent is trying to predict. The more micro-simulations the agents run, the more likely they are to find the optimal decision. The longer they run the micro-simulations for, the more they will consider the further consequences of their actions.

This process is generally similar to using Monte-Carlo sampling to approximate solutions to partially observable Markov decision processes (Lovejoy 1991). However, in this case, the Markov chain is being defined by an agent-based model. This offers a natural way to define what would otherwise be very complex processes. It also allows researchers to directly inspect what is happening in the agent's cognitive model, and understand why the agent acts the way it does. This also demonstrates the method's advantage over, for instance, using neural networks to guide agents' actions as in paragraphs 80-85. Neural networks are infamously difficult to interpret. In the micro-simulation approach, the situations which the agent considers when making its decisions may be directly observed and meaningfully interpreted.

Dimensions of LevelSpace Model Relationships

Multi-Level Agent-Based Modeling (ML-ABM) is an emerging approach that will be used in by modelers in both (social) sciences and in education. For the purpose of creating a common ground for talking about ML-ABM, modelers and researchers need a common vocabulary for describing these larger systems of *models*. In particular, having a descriptive language will help in *communicating*, *designing*, *verifying* and *validating* ML-ABM systems.

One obstacle to a common vocabulary for ML-ABM is that agent-based modeling is used to describe a variety of approaches that differ widely — conceptually, linguistically, and execution-wise — in how they allow modelers to practically design and code models. These differences present modelers with specific constraints and affordances[10] when building ML-ABM systems, and consequently also with different conceptual problems to solve and describe.

For this reason, a vocabulary of the specifics of ML-ABM will depend on the modeling framework and general approach to modeling. In the following, we propose six different dimensions along which relationships between ML-ABM models created with LevelSpace might vary. While these *may* apply to other ML-ABM approaches or languages, we have developed this set of dimensions specifically for better describing ML-ABMs built with LevelSpace.

Homogenous vs. heterogeneous relationships

In LevelSpace systems, many models run at the same time. Sometimes these models are based on the same .nlogo file, and other times those models are based on different .nlogo files. We therefore need to make a clear distinction between what we call a *template model* – the model file, not yet instantiated – and an *instance* of that model, containing agents, environments, etc. In describing the relationships that constitute LevelSpace systems, an important distinction is whether they take place between two models that are instances of the same template, or whether they take place between models that are instances of different templates. We call the former *homogenous* model relationships, and the latter *heterogeneous* model relationships. While we did not include any examples of homogeneous model relationships, in other research we built an epidemiological ML-ABM in which instances of the same template model each represent different countries or other geographical regions (Vermeer et al. 2016). Such ML-ABM systems could represent geography by "tiling" copies of a single template model to represent contiguity, or by linking them in a network, to represent connectivity through air travel or other transportation networks.

Unidirectional vs. bidirectional relationships

Directionality refers to whether information in one model affects the behavior of agents in the other. In our examples, all participating models inform and influence each other bidirectionally, but one could imagine a ML-ABM system in which information is passed unidirectionally from one model to another. In fact, in our first prototype of the WSP-CC model system, only the Climate Change model affected grass regrowth time but with no reciprocal effects. We therefore think it is important to make a distinction between the directionality, respectively unidirectional and bidirectional, of information passing between a dyad of models.

Persistence

In the Ecosystems and Climate Change-example, both the WSP and the CC model are opened at the beginning of the model run, and both models remain open throughout the run. In contrast, in the Wolf Chases Sheep-example, one model is opened, used to resolve the sheep chase, and then discarded. Finally, in the Wolf Sheep Brain-example, models are opened and closed with the births and deaths of wolves and sheep. We see the distinction between these three different kinds of

persistence as valuable for describing and classifying LevelSpace systems: *system-level persistence*, in which models are opened at the beginning of the model run and closed at the end; *agent-level persistence*, in which each child model shares its lifespan with a particular agent; and *event-level persistence*, in which child models are opened in response to a particular event or state of the world and closed when that event has been resolved.

Time synchronicity

The fourth dimension describes how time is modeled relatively between two model-instances. In the Ecosystems and Climate Change-example, the WSP and the CC model run in tandem from the moment they are set up. Similarly, in our Agent Cognition examples, each brain model is run in tandem with WSP. In contrast to these two, in the Wolf Chases Sheep-example, the WSP model pauses until the outcome of the wolf-sheep chase has been evaluated. These differences point to an important distinction among model-relationships: those that are *synchronous*, and those that are *asynchronous*.

One complication is that time may be modelled differently in different models due to differences of scale. For instance, in the Ecosystems and Climate Change-example, the CC model runs at a much slower rate. To compensate for this, we call the GO-procedure 10 times for each time we call the GO-procedure in WSP. However, we would still consider this a synchronous relationship because the *ratio* between their time steps remain constant. In other words, a model dyad is synchronous if the *ratio* between the models' time steps is constant, and otherwise it is asynchronous.

Hierarchy

In the Wolf Chases Sheep-example and the Agent Cognition-example, the chase and brain models are sub-systems of the larger phenomenon in the other model. For instance, the wolf and the sheep in the Wolf Chases Sheep model also exist in the Wolf Sheep Predation model. Similarly, each of the Brain models "belongs" to a wolf or a sheep in the WSP model. In contrast, in the Ecosystems and Climate Change-example, neither model can be said to be a sub-system of the other. The distinction between hierarchical and non-hierarchical model-relationships is important, and is resolved by asking whether the phenomenon in each of the models can be seen as respectively a sub- or super-system of the other[11].

Model-Ratios

In our first two examples – Ecosystems and Climate Change, and Wolf Chases Sheep, respectively - we *need* just one instance of each template model at the same time: respectively, one instance of WSP and one of CC, and one instance of WSP and one of WCS[12]. However, in our third example, we have one instance of WSP and many instances of the Brain model. Consequently, making a distinction between the number of models in a relationship is an important part of a multi-level model systems typology. The model ratio is often implied by the hierarchy of which is it part: if you have an agent-level hierarchical model, you will most likely have a one-to-many model-ratio. However, we can see the one-to-many ratio not just at the model-level, but at the agent-level too: consider a LevelSpace system of an industry, where agents represent organizations that open and close different departments in an ad-hoc manner, and where each department is represented by a child model. This ML-ABM system would contain a one-to-many model-relationship between an organization and its departments at the agent-level. Conversely, consider a model system in which we model hundreds of cities, each producing greenhouse gases. These could all contribute greenhouse gases to one climate change model. In that case, there would be a many-to-one model-ratio between city-models and the climate change model.

Towards a Typology of Multi-Model Dyadic Relationships

We have presented six dimensions that we believe address important aspects of the dyadic relationships and interactions between model *templates*. See Table 1 for a full overview. Between the six dimensions, there are 96 combinations of possible dyadic relationship descriptors in total. This raises interesting questions about what kinds of combinations and dyadic relationships will be more or less common and more or less generative for modeling phenomena in different fields. Our future work will focus on understanding these combinatorics with an aim to map how particular combinations of these six dyadic relationship dimensions can best be used to solve specific modeling problems.

Table 1: The Six Dimensions of Dyadic Model Template Relationships

Dimension	Meaning	Potential values
Model Similarity	Are the interacting models instances of the same template?	Homogeneous / Heterogeneous
Directionality	Are both models affecting each other, or is only one model affecting the other?	Unidirectional / bidirectional
Persistence	What is the lifespan of one model relative to the other?	System-level, agent-level, event-level
Synchronicity	Is time linearly correlated between instances of the two templates?	Synchronous / asynchronous
Hierarchy	Is the modeled phenomenon of one component of the ML-ABM a subsystem of the other?	Hierarchical / non-hierarchical
Model-ratios	Are there more than one instance of a template per one instance of the other template?	1:1, 1:many

Some combinations may be rare, and other combinations may even be conceptually contradictory. We want to mention that it is not obvious to us which are which. When writing this paper, the authors discussed some combinations that we thought would be unlikely. One of these was the homogeneous, agent-persistent dyadic relationship: a model that opens a copy of itself whose persistence is determined by one single agent. At first this seemed unlikely, but as we showed with the latter of the two Agent Cognition examples, we did in fact use an instance of the WSP model to run micro-simulations that was agent-persistent and that was an instance of the WSP model. To us, this illustrated how our inability to envision this case in advance was simply a shortcoming of our imagination and intuition for which of these combinations are viable and useful.

Discussion and Conclusion

Multi-Level Agent-Based Modeling (ML-ABM) is a new approach that offers exciting opportunities for new ways of modeling connected systems. It also raises interesting questions, some of which we addressed in this paper. One focal question that we addressed is: 'what can ML-ABMs be used for'? Following Morvan's (2012) three use cases for ML-ABM, we demonstrated how LevelSpace can address each of them. Specifically, we showed how LevelSpace is versatile and allows modelers to (1) connect related phenomena through heterogeneous coupling; (2) "zoom in" on a particular event in a multi-level system; and finally, (3) simultaneously model processes at different scales.

Further research is needed to more generally classify modeling use cases and the ML-ABM architectures that can address them.

An important affordance of the design of LevelSpace is how easy it is to connect existing models, owing to the fact that LevelSpace works by opening NetLogo models from inside other NetLogo models without necessarily making changes to the code of those original models. This provides an easy way to expand on, or contest assumptions in existing models. As an illustration of this, all three of our model examples use either the NetLogo Models Library version, or a very slightly modified version, of the NetLogo Wolf Sheep Predation Model (Wilensky 1997).

Our examples also showed how the LevelSpace programming primitives align with the existing NetLogo language and illustrated how easily models can be connected with human readable code. Given the large number of existing, peer-reviewed studies using NetLogo, LevelSpace provides opportunities for collaborative modeling by combining forces, *and models*, with relatively minor effort. This raises interesting questions—both *practically*, about how to push the modeling research agenda in new and more collaborative directions, and *scientifically*, about how we can use the process of combining models to critique and validate our existing models.

The last question that we addressed is how to conceptualize ML-ABM in order to create a shared vocabulary. We presented six dimensions by which relationships between models in LevelSpace can be described and distinguished in a more fine-grained manner. In future work, we will explore the combinatorial space of these six dimensions in an effort to map out the viability of particular combinations, and to investigate how some may be particularly well suited for addressing particular modeling problems or patterns.

Finally, while we did not spend much time discussing this here, LevelSpace raises an interesting set of questions relating to the educational potential of ML-ABM. There is already a long tradition of improving students' causal-reasoning through learning with ABMs (Wilensky & Jacobson 2014), particularly with NetLogo due to its "low threshold, high ceilings"-approach. This work has leveraged ABM's ability to decompose complex phenomena into their constituent parts, and the cognitive benefits that this gestalt gives to understanding complex causality. Our own early work (Hjorth et al. 2014) suggests that being able to model systems as consisting of smaller but connected systems helps learners more easily connect causality across many levels. In our future work, we will further explore the ways in which ML-ABM can help learners make sense of multi-level complex systems and phenomena.

LevelSpace joins a recent but rapidly growing effort to offer multi-level agent-based modelling platforms. NetLogo has been embraced by the modeling community for decades, and the addition of ML-ABM capabilities has exciting prospects. We offered three examples as a proof of concept of how to build ML-ABM models with NetLogo and LevelSpace, and showed how problems that LevelSpace can solve lie within the established set of ML-ABM modeling problems. We are excited to invite the community to join us in using LevelSpace to build new, interesting models, explore novel modeling techniques, and to expand on modeling concepts and theory in the process.

Acknowledgements

This work is supported in part by the National Science Foundation under NSF grant 1441552. However, any opinions, findings, conclusions, and/or recommendations are those of the investigators and do not necessarily reflect the views of the Foundation.

Notes

1. Note that the following is not meant to be a comprehensive list of existing work, but rather give a general idea of what approaches have been tried. See Morvan (2012) for a more comprehensive review of approaches, languages and frameworks.
2. https://github.com/CRESS-Surrey/eXtraWidgets.
3. https://github.com/netlogo/levelspace.
4. https://github.com/NetLogo/Population-Dynamics-and-Climate-Change.
5. Albedo is a measure of the amount of light that is reflected vs the amount of light that is absorbed by a particular surface.
6. https://github.com/NetLogo/Wolf-Chase-Sheep-JASSS-Version.
7. We open and close the model here only to illustrate where one could do it. In the GitHub version of this model, we open the model only once, and simply "reuse" the same model. This makes the model run considerably faster.
8. https://github.com/NetLogo/Sheep-with-brains.
9. https://github.com/NetLogo/Wolf-Sheep-Predation-Micro-Sims.
10. We mean 'affordance' in the design-sense of the word: that it allows – and even invites - a particular set of actions or manipulations while making other actions more difficult or even impossible.
11. To avoid confusion, we want to emphasize that considerations about hierarchy do not refer to the LevelSpace "family hierarchy" of models, i.e. parents, children, grand-children, etc. Rather, it speaks conceptually to the modeled phenomena in the models. For instance, one model could load instances of two different models without either of those child models being hierarchically related to their parent in a conceptually meaningful way. Similarly, these two child models can be in a sub-system-super-system relationship to each other, even though neither is each other's parent or child model in the LevelSpace hierarchy.
12. We emphasize 'need' for an important reason: there are many ways of designing a model system, and many of them will be behaviorally equal. In constructing these typological dimensions, the question we have to ask is, what is *needed* in order to preserve the necessary information. In the case of the Wolf Chases Sheep model, it would be possible to load a model for each "meeting" between a wolf and a sheep at the same time, thus changing the model ratio to a one WSP to many WCS models. However, because each event is resolved entirely before the next event is resolved, no information needs to be preserved in the state of the model, and we therefore only *need* one WCS model for the system to function.

References

BAR-YAM, Y. (1997). *Dynamics of Complex Systems* (Vol. 213). MA: Addison-Wesley Reading.

BERLAND, M. & Wilensky, U. (2005). Complex play systems: Results from a classroom implementation of VBOT. W. Stroup and U. Wilensky (Chairs) & CD Lee (Discussant), Patterns in group learning with next-generation network technology. *The Annual Meeting of the American Educational Research Association 2005*, Montreal, Canada.

BERLAND, M. & Wilensky, U. (2006). Constructionist collaborative engineering: Results from an implementation of PVBOT. Annual Meeting of the American Educational Research Association 2006, San Francisco, CA.

BLIKSTEIN, P & Wilensky, U. (2007). Bifocal modeling: a framework for combining computer modeling, robotics and real-world sensing. Annual Meeting of the American Educational Research Association 2007, Chicago, IL.

BLIKSTEIN, P & Wilensky, U. (2008), Implementing multi-agent modeling in the classroom: Lessons from empirical studies in undergraduate engineering education. Proceedings of the International Conference of the Learning Sciences 2008. Utrecht, The Netherlands.

BLIKSTEIN, P & Wilensky, U. (2010). 'MaterialSim: A constructionist agent-based modeling approach to engineering education.' In M. J. Jacobson & P. Reimann (Eds.), *Designs for Learning Environments of the Future: International Perspectives from the Learning Sciences*. Boston, MA: Springer, pp. 17-60. [doi:10.1007/978-0-387-88279-6_2]

CENTOLA, D., Mckenzie, E., Wilensky, U. (2000). Survival of the groupiest: Facilitating students' understanding of multi-level evolution through multi-agent modeling - The EACH Project. Proceedings of The Fourth International Conference on Complex Systems. New Hampshire, New England.

DICKERSON, M. (2015). Agent-based modeling and NetLogo in the Introductory Computer Science curriculum: Tutorial presentation. *Journal of Computing Sciences in Colleges*, 30(5), 174–177.

DROGOUL, A., Amouroux, E., Caillou, P., Gaudou, B., Grignard, A., Marilleau, N., Taillandier, P., Vavasseur, M., Vo, D. A., Zucker, J.D. (2013). Gama: multi-level and complex environment for agent-based models and simulations. Proceedings of AAMAS '13. International Foundation for Autonomous Agents and Multiagent Systems. St. Paul, Minnesota. [doi:10.1007/978-3-642-38073-0_25]

EPSTEIN, J. M. (1999). Agent-based computational models and generative social science. *Complexity*, 4: 41-60. [doi:10.1002/(sici)1099-0526(199905/06)4:5<41::aid-cplx9>3.0.co;2-f]

EPSTEIN, J. M. (2006). *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton, NJ: Princeton University Press. [doi:10.23943/princeton/9780691158884.003.0003]

EPSTEIN, J. M., & AXTELL, R. L. (1996). *Growing Artificial Societies: Social Science from the Bottom Up* (1st ed.). Washington, D.C: Brookings Institution Press. [doi:10.7551/mitpress/3374.001.0001]

GILBERT, N. (2008). *Agent-Based Models*. London, UK: Sage.

GILBERT, N., & Terna, P. (2000). How to build and use agent-based models in social science. *Mind & Society*, 1(1), 57–72. [doi:10.1007/bf02512229]

GUO, B. & Wilensky, U. (2018). Mind the gap: Teaching high school students about wealth inequality through agent-based participatory simulations. Proceedings of Constructionism 2018. Vilnius, Lithuania.

HAUKE, J., Lorscheid, I., Meyer, M. (2017). Recent development of social simulation as reflected in JASSS between 2008 and 2014: A citation and co-citation analysis. *Journal of Artificial Societies and Social Simulation*, 20(1), 5: http://jasss.soc.surrey.ac.uk/20/1/5.html. [doi:10.18564/jasss.3238]

HEAD, B. & Wilensky, U. (2018). Agent cognition through micro-simulations: Adaptive and tunable intelligence with NetLogo LevelSpace. *Proceedings of Ninth International Conference on Complex Systems 2018*. Cambridge, Massachusetts. [doi:10.1007/978-3-319-96661-8_7]

HJORTH, A., Brady, C. & Wilensky, U. (2018). Sharing is caring in the Commons: Students' conceptions about sharing and sustainability in social-ecological systems. Proceedings of Constructionism 2018. Vilnius, Lithuania.

HJORTH, A., Head, B. & Wilensky, U. (2015). LevelSpace NetLogo Extension. http://ccl.northwestern.edu/levelspace. Evanston, IL: Center for Connected Learning and

Computer-Based Learning. [doi:10.1109/wsc.2015.7408430]

HJORTH, A. & Wilensky, U. (2014). Redesigning your city: A constructionist environment for urban planning education. *Informatics in Education*, 13(2), 197. [doi:10.15388/infedu.2014.02]

HJORTH, A. & Wilensky, U. (2018). Urban planning-in-pieces: A computational approach to understanding conceptual change and causal reasoning about urban planning. Proceedings of Constructionism 2018. Vilnius, Lithuania.

LEVY, S. & Wilensky, U. (2005). An analysis of students' patterns of exploration with NetLogo models embedded in the connected chemistry environment. Proceedings of the annual meeting of the American Educational Research Association 2005. Montreal, CA.

LOVEJOY, W. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28(1), 47–65. [doi:10.1007/bf02055574]

MARCENAC, P. & Giroux, S. (1998). Geamas: a generic architecture for agent-oriented simulations of complex processes. *Applied Intelligence*, 8(3), 247–67.

MAUS, C., RYBACKI, S. & UHRMACHER, A. (2011). Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology*, 5(1), 1. [doi:10.1186/1752-0509-5-166]

MICHEL, F. (2007). The IRM4S model: The influence/reaction principle for multiagent based simulation. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems* pp. 1-3)ì. [doi:10.1145/1329125.1329289]

MINAR, N., Burkhart, R., Langton, C. & Askenazi, M. (1996). The Swarm simulation system: a toolkit for building multi-agent simulations. Tech. Rept., Santa Fe Institute, Santa Fe, NM.

MITCHELL, M. (2009). *Complexity: A Guided Tour*. Oxford University Press: Oxford, UK.

MORVAN, G. (2012). Multi-level agent-based modeling: a literature survey. *ArXiv* Preprint ArXiv:1205.0561.

MORVAN, G. & Kubera, Y. (2017). On time and consistency in multi-level agent-based simulations. *ArXiv* Preprint ArXiv:1703.02399.

MORVAN, G. Veremme, A. & Dupont, D. (2010). IRM4MLS: The influence reaction model for multi-level simulation. *International Workshop on Multi-Agent Systems and Agent-Based Simulation*. Heidelberg: Springer. [doi:10.1007/978-3-642-18345-4_2]

RAILSBACK, S., Allyón, D., Berger, U., Grimm, V., Lytinen, S., Sheppard, C. & Thiele, J. (2017). Improving execution speed of models implemented in Netlogo. *Journal of Artificial Societies and Social Simulation*, 20(1), 3: http://jasss.soc.surrey.ac.uk/20/1/3.html. [doi:10.18564/jasss.3282]

SCERRI, D., Drogoul, A., Hickmott, S. & Padgham, L. (2010). An architecture for modular distributed simulation with agent-based models. Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. Budapest, Hungary.

SCHELLING, T. (1969). Models of segregation. *The American Economic Review*, 59(2), 488–93.

SENGUPTA, P. & Wilensky, U. (2009). Learning electricity with NIELS: Thinking with electrons and thinking in levels. *International Journal of Computers for Mathematical Learning*, 14(1), 21–50. [doi:10.1007/s10758-009-9144-z]

SOLOYEV, A. Mikheev, M., Zhou, L., Dutta-Moscato, J., Ziraldo, C., An, G., Vodovotz, Y. & Mi, Q., (2010). SPARK: A framework for multi-scale agent-based biomedical modeling. Proceedings of the 2010

Spring Simulation Multiconference. Orlando, Florida. [doi:10.1145/1878537.1878541]

STIEFF, M. & Wilensky, U. (2003). Connected chemistry—Incorporating interactive simulations into the chemistry classroom. *Journal of Science Education and Technology*, 12(3), 285–302.

STONEDAHL, F., Wilkerson-Jerde, M. & Wilensky, U. (2009). Re-conceiving Introductory Computer Science Curricula through agent-based modeling. Proceedings of the AAMAS 2009 Workshop on Educational Uses of Multi-Agent Systems. Budapest, Hungary. [doi:10.4018/978-1-60960-080-8.ch001]

THIELE, J., Winfried, K. & Grimm, V. (2012). Agent-based modelling: Tools for linking NetLogo and R. *Journal of Artificial Societies and Social Simulation* , 15(3), 8: http://jasss.soc.surrey.ac.uk/15/3/8.html. [doi:10.18564/jasss.2018]

TINKER, R. & Wilensky, U. (2007). NetLogo Climate Change Model. http://ccl.northwestern.edu/netlogo/models/climatechange. Center for Connected Learning and Computer-Based Modeling, Evanston, IL: Northwestern University.

TISUE, S. & Wilensky, U. (2004). NetLogo: A simple environment for modeling complexity. International Conference on Complex Systems. Boston, Massachusetts.

VERMEER, W., Head, B. & Wilensky, U. (2016). The effects of local network structure on disease spread in coupled networks. International Workshop on Complex Networks and their Applications. Milan, Italy. [doi:10.1007/978-3-319-50901-3_39]

WAGH, A. & Wilensky, U. (2012). Evolution in blocks: Building models of evolution using blocks. Proceedings of Constructionism 2012. Athens, Greece.

WAGH, A. & Wilensky, U. (2013). Leveling the playing field: Making multi-level evolutionary processes accessible through participatory simulations. *Proceedings of the Biannual Conference of Computer-Supported Collaborative Learning (CSCL)*. Madison, Wisconsin.

WAGH, A. & Wilensky, U. (2014). Seeing patterns of change: Supporting student noticing in building models of natural selection. Proceedings of Constructionism 2014. Vienna, Austria.

WILENSKY, U. (1997). NetLogo Wolf Sheep Predation Model. http://ccl.northwestern.edu/netlogo/models/wolfsheeppredation. Center for Connected Learning, Northwestern University. Evanston, IL.

WILENSKY, U. (1999). NetLogo. Evanston, IL: Center for Connected Learning and Computer-Based Modeling. Northwestern University.

WILENSKY, U., & Jacobson, M. J. (2014). 'Complex systems and the learning sciences.' In R. K. Sawyer (Ed.), *The Cambridge Handbook of the Learning Sciences*, Second Edition. Cambridge University Press, pp. 319-338. [doi:10.1017/cbo9781139519526.020]

WILENSKY, U., & Rand, W. (2015). *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. New York: MIT Press.

WILENSKY, U. & Reisman, K. (2006). Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing computational theories—An embodied modeling approach. *Cognition and Instruction*, 24(2), 171–209. [doi:10.1207/s1532690xci2402_1]