Generalizing the Random Copying Model for Cultural Transmission Research [*]

Mark E. Madsen
Department of Anthropology, University of Washington
(`madsenm@u.washington.edu`*)*

February 2007

Abstract. Bentley et al. 2007 present a simple null model for imitation in cultural populations. This "random copying model" attempts to provide a simple generative model for the imitation of "neutral" traits within a well-mixed population, with some innovation/invention rate for new traits, and stochastic loss of low-frequency traits by drift. The current document outlines a series of generalizations and refactorings of the basic agent-based model, suitable for additional experimentation and analysis of cultural null models as well as various types of biased copying. The ABM itself is written in Repast 3.x, a popular Java-based agent modeling toolkit with good documentation, available on open-source license terms for all common platforms.

Keywords: cultural transmission, cultural evolution, agent-based modeling, RePast 3.x

1. Introduction

In their 2007 article, "Regular rates of popular culture change reflect random copying," Alex Bentley, Carl Lipo, Harold Herzog, and Matthew Hahn present a simple model of random cultural imitation in a well-mixed population. This model posits a null hypothesis for the cultural transmission of "neutral" traits, which is that nothing more complex than random copying of a fellow population member's trait is needed to account for the patterns seen at the population level.

In what follows, I focus not on the scientific aspects of the random copying model, but the software engineering aspects of generalizing the Bentley et al. model for additional cultural transmission experiments. Bentley et al.'s model (hereafter referred to as the RCM or random copying model) is written in Java, using the Repast agent-based simulation toolkit. With suitable generalization and refactoring, the RCM will serve as a useful "platform" model for both extending Bentley et al.'s analysis, but performing a wide variety of computational "experiments" of cultural transmission.

[*] This work is licensed under the Creative Commons NonCommercial-Attribution-ShareAlike License version 2.5 (BY: $ ⓒ). To view a copy of this license, visit http://creativecommons.org or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

I began working on generalizing the Bentley et al. model because it is clean, simple, and reasonably generic "out of the box." Starting from scratch allows one to design without history or constraints, but it is often valuable to start from something which works, and modify it incrementally. I chose the latter approach, incrementally evolving the model's structure. In the next several sections I outline the reorganization, refactoring, and generalization of the simulation codebase I'm performing.

This document is a work-in-progress, and will be updated as we continue to develop and use the transmission framework for new research projects.

2. Modeling Philosophy

One of the hardest aspects of using agent-based models in scientific research is translating the theoretical "model" into the simulation realm. In my past experience with simulations written in Swarm (Objective C), Repast (Java), and from scratch in C++, minor "bugs" are the least of one's worries. Even a relatively simple simulation is a complex body of code, and ensuring the correct timing and ordering of agent updates, interactions, and observations in order to measure what we *believe* we're measuring can be quite difficult.

In generalizing the Random Copying Model, I began with the hypothesis that in most agent-based models designed to study population-level "inheritance" problems, several aspects of the model can be kept strictly orthogonal in the design of the simulation model. And if the code can be kept as separate as possible, each "facet" of such a model can be tested rigorously with simple, "null" cases. Schedules and event interleaving can be tested without complex activity getting in the way. Data collection, displays and analysis can be tested on known data sets. Agent interaction can be tested in isolation, knowing that data collection or other aspects of the model aren't affecting that interaction via side effects.

Of course, the price of all this cleanliness and separation is a more complex body of code. What follows are notes on the current version of the Random Copying Model, and the changes I'm trying in order to achieve a model which really lives up to the promises of the previous paragraph. These notes will also serve as partial documentation for using and extending the model once released.

3. Basic Reorganization and Runtime Updates

For ease of constructing many models from the basic codebase, I reorganized the model into a series of java packages under a “src” directory. Packages for this model are located under `org.mmadsen.sim.transmission` and comprise a functional decomposition of the code:

Package	Function
agent	Classes implementing alternative types of agents
analysis	Classes implementing data collection or analysis upon a model
interfaces	Generic interfaces general model interrelationships
models	Classes which extend `SimModelImpl` and define a given model
population	Classes which implement agent populations
rules	Classes which implement transmission rules
test	JUnit test classes and test harnesses
util	Helper and utility classes

4. Refactoring Notes

At a high level, our goal is to create a framework for examining the dynamics of many different cultural transmission models and scenarios, with agents of varying internal complexity, different types of adoption and imitation rules (e.g., random, imitate most frequent, imitate the successful), and differing population structure (e.g., well-mixed, lattices, and social network graphs of varying topology).

We also seek an easy way to perform experimental measurements of these dynamics, at both individual and population levels. The platform should allow new types of measurements to be “plugged in” to the basic model framework, and given only knowledge of agent and model public methods, the plug-in should manage its own data collection, statistical analysis, displays and graphs, and data persistence. Code for data collection and visualization should not be mingled with model code, and the former must use clean public APIs to access the latter. This philosophy promotes modularity, simple extension of existing transmission models to look at new measurements and types of analysis, and helps ensure the stability and accuracy of the model core once it is stable and the bug count is low.

4.1. Refactoring Model and Measurement

The first separation I performed was to extract data collection code from the core model class (named `RandomCopyingModel`). The current design defines a single interface for measurement and data collection classes (named `IDataCollector`). This interface defines a simple four-method contract which defines a data collection "cycle."

In addition, each `IDataCollector` class carries a "type code," which allows the model to keep a map of which data collectors are running by type, and easily access them at any time. This is useful for removing or disabling data collection based upon GUI settings, or altering the schedule for data collection at runtime. For example, we might want data snapshots of the running model gathered, but only at specific points in the simulation run, and we don't want to pay the performance penalty of scheduling the snapshot code when it's not needed. In the present version of the model, the "type code" for an `IDataCollector` instance is the `getClass().getSimpleName()`, and does not currently distinguish between identically named classes in different packages or between object instances of the same class, if they perform different data collection functions. This needs to be evolved to guard against such issues.

The model class stores a `List<IDataCollector>`, to which all data collection objects should be added. The simulation author, within the Repast `setup()` method, should instantiate any `IDataCollector` classes, call their `build()` methods (to allow setup and reset to occur), and add the objects to the data collector list. The objects should also be added to a `Map<String, IDataCollector>` which stores the data collector instances under the "type code" key discussed above. The simulation author is also responsible for inserting the object instances into the data collector map using `getDataCollectorTypeCode()` to retrieve the collector object's type code.

The model class then takes care of initialization, in the Repast `begin()` method, by iterating over the data collector list, calling `initialize()` on each data collection object. We separate object construction and initialization because Repast does so within models, so that the simulation can run and display the basic GUI to gather parameters, which are then available by the time the user clicks the "begin" button. Thus, in the `begin()` model method, we allow each data collector to query the model for parameters relevant to its own configuration and operation.

In the current version, scheduling of data collection is very simplistic, and not well refactored. Currently, the model's Action methods (`mainAction` and `initialAction`) each iterate over the list, calling the `process()` method. If a given data collector is not supposed to run on each model tick, as with data file recording in the original Random Copying Model, the data collector itself is responsible for detecting this and returning from a call to `process()` without performing any action.

This is clearly not a clean design; ultimately each data collector object will be able to add itself to a schedule which will call `process()` at the correct times and intervals, and the current list iteration will be removed from the Action methods. (Note: as of version 1.2, the data collector objects are added to an ActionGroup which is scheduled, with a BasicAction method for each `process()` method. This still doesn't handle more complex scheduling details, but I'm going to leave that aside in 1.3 and work on modularity of transmission and other "model" rules in preparation for spatial structure.

4.2. Inter-Module Data Sharing

4.3. Data Collector Refactorings in the Bentley Model

Given this structure, the turnover graph and data snapshots in the original Random Copying Model were moved into separate classes: `top40DataFileRecorder` and `TurnoverGraphCollector`. The latter refactoring, in particular, removes a great deal of code and complexity from the model class. The inner class `Turnover` moves to the `TurnoverGraphCollector`, and at some point the calculation of sorted "top40" lists will as well, since the latter constitute *analysis* of a transmission model (random copying) rather than part of the agent interaction itself.

As part of this refactoring, I discovered that the original methods of calculating turnover within the model fail under certain circumstances. This isn't an issue for Bentley et al. 2007, because the data processing for that article was performed outside the Repast context (Bentley, personal communication). But in extending the analysis and models to more complex cases, it would be good to have analysis modules which could reduce the amount of data post-processing required.

One "programmers" note about the refactorings described here may help others extend the model later. It's taken a bit of work to get everything working correctly and cleaning up after itself, mostly because Repast is still a fairly low-level modeling framework (Repast Simphony promises to change this dramatically). Repast, like Swarm, isn't particularly good about separating initial invocation of the model and subsequent runs of the model from the same invocation (e.g., by hitting the stop and reset buttons in the GUI, or multiple batches, etc). In particular, by putting the graph and its collection class into a separate object, I ended up with double construction and then null pointer errors unless the `setup()` method in the model kept good track of whether we'd already been through the loop or not. This is the origin of the `completion()` method in the `IDataCollector` interface: it allows each data collector to clean itself up, dispose of GUI windows and resources, close file references, either at the end of a simulation or when the simulation is reset for a subsequent run. It's all working nicely now with no apparent memory leakage, but the mechanics aren't very pretty. Fortunately the code to track this sort of thing is in the Model class, and Agents, IDataCollectors, or future simulation "plug-in" types will have to know anything about it.

4.3.1. *Example Data Collector: TraitFrequencyAnalyzer*

I added `TraitFrequencyAnalyzer`, written from scratch as a "paradigm" example of how a modular data collection and analysis system would work. The class accesses *only* the agent list held by the model, caches its own "top N" lists between model ticks, holds its own graph instances, and performs all turnover calculations. Thus, it can be "turned off" cleanly if desired, and it does not inject any code into the model itself. I describe this class in some detail in this section, as a guide to writing additional data collection and analysis modules in the future.

Little initialization is required in the `build()` method for this class; we instantiate a map for frequencies that will hold instances of an inner value class (`TraitCount`), indexed by the agent's trait identifier (here, an integer). The `completion()` method is similarly very simple, responsible only for ensuring disposal of any graphs or other GUI elements. The `initialize()` method similarly does little work, in this case constructing two `OpenSequenceGraph`s when the user hits the "begin" button on a simulation. One graph displays "top N" turnover over time, and the second displays the total number of variants present in the population throughout the simulation (with the pure random copying model and no additional rules or structure, this converges on a mean value of 4Nmu), but since the population is finite the value fluctuates around this infinite-limit value.

All action in this data collector is driven by the `process()` method, called once per model tick by the model's scheduled "main action." The first task is to refresh our reference to the model's agent list, since we can make no assumptions that it hasn't been altered (agents removed or added) since the previous tick. We then clear out the internal frequency map to count traits afresh, and clear out an internal list of `TraitCount` objects which will be filled later.

Trait counting is done in a single pass through the agent list, using the Java equivalent of a functional programming style. By this I mean that we iterate over the list, and apply a "functor" (function object) to each element. This is very similar to the style of programming familiar from Perl, Ruby, or Lisp and is very efficient and clean. This is done in (current versions of) Java by creating an inner helper class that implements the `Closure` interface from Jakarta Commons Collections, and then passing the agent list and the closure class to the Commons Collections helper method `CollectionUtils.forAlldo()`. The closure class here is `FrequencyCounter`, and in its `execute()` method it checks if the passed `RCMAgent`'s trait is already indexed in the trait frequency map, calls the `TraitCount.increment()` method if so, and if not, constructs a new `TraitCount` object for the trait and inserts it into the map.

Once the counting pass is complete, the frequency map contains `TraitCount` objects representing the frequency of each trait in the agent population. We then obtain a reverse-sorted list of these traits by frequency by having `TraitCount` implement the `Comparable` interface and define a `compareTo()` method which provides a "natural" sort order based on the count, not the trait ID or object hash code, and reverses the ordering to obtain a descending order sort. Given this comparator method, the standard Java `Collections.sort()` method returns a list already sorted with highest frequency first, lowest last. This seems like a lot of machinery to do a sorting, but we essentially get a "top N" list for free, by simply reading the first N items off this sorted list.

To facilitate calculating turnover, we keep two versions of this sorted `TraitCount` list: a cached copy from the previous model tick and the list being constructed for this model tick by `process()`. Presumably we could extend this by keeping the data from all model ticks if desired, or persisting it to a database for analysis.

Finally, the `process()` method calls the `step()` method on the "turnover" and "variability" graphs. The graphs themselves are responsible for actually calculating turnover, much as in the original model. I use the same structure as the original model: a `TurnoverSequence` class which provides the graph with the latest turnover value at each `step()` of the graph; similiarly, a `TotalVariabilitySequence` provides that graph with a stream of `size()` values for the current set of sorted TraitCount objects.

I calculate turnover slightly differently than the original model, in an attempt to avoid those situations where there are less than 40 traits in the population (which caused anomalies given fixed arrays since random data were being incorporated into the analysis if the array wasn't filled all the way with "real" data). In this class, I define: *turnover is the number of elements in two sets (previous top N and current top N) that are not present in the intersection of the two sets.*. This definition counts traits that are present in the previous list, but not present in the current list, as well as the reverse situation.

The equation is thus: `turnover = (prevSortedTraitCounts.size() + curSortedTraitCounts.size()) - ( 2 * intersection.size());`. Intersection is performed on two temporary lists of the pure sorted trait IDs, passed to the Commons Collections utility method `CollectionUtils.intersection()`.

If the lists are larger than 40 (really, a configurable parameter), we trim the bottom of trait lists before performing the turnover calculation. This means that all of the original frequency counts are available to other methods, and "top N" restriction is only meaningful to this particular sequence class and graph.

4.4. Plug-In Parameterization

One thing that I'm still thinking through is how to deal with the Repast "model parameter" subsystem and the notion of a plug-in architecture. The difficulty here is that the plug-in class (e.g., a class which implements `IDataCollector`) ought to contain its own parameterization, which it then contributes to the model at setup and model start. Right now, things are a bit of a hybrid: `getInitParams()` is called so early in initialization that my design for constructing plug-ins hasn't executed yet. So I'm tracing out how `getInitParams()` is called by `SimInit` and dependent classes for initialization, to see what we can do.

My ideal design would be to have some way to say "run Model A, and do it with analysis plug-ins X and Y." If I can't get there, you'll probably still have to do some explicit loading and initialization of plug-in classes, but we could perhaps use parameter files to let each plug-in class contribute parameters that appear in the initial GUI.

NOTE: Still working on this (2/28/07) but am having some luck - in the SimModelImpl constructor you can add PropertyDescriptors dynamically, like I do for the initial trait structure pull-down.

4.5. Refactoring Model and Inter-Agent Interaction

4.6. Isolating Population Structure

Ultimately, my goal for agent populations is to allow other aspects of the model (e.g., interaction rules, data analysis) to be orthogonal from the way agents are arrayed in a population. This will allow models to be held "constant" and their results compared when the population is differently structured. For example, one might want to compare a well-mixed population baseline against structures like regular lattices or various types of network models.

With this goal in mind, I defined an interface `IAgentPopulation` which represents an abstract agent population. Each instance of `IAgentPopulation` is created by a "factory" class, which must implement the contract specified in `IPopulationFactory`. The point of this double abstraction is that different types of populations might require different logic in their factory classes for construction, and creating small units of encapsulation helps us avoid buggy, hard-to-understand factory classes. Since the factory classes all follow the same "contract," however, we can configure a model class easily to load any factory and thus potentially construct any type of population. One way this can be made dynamic is for each factory to export an XML descriptor of the population classes it is capable of constructing, perhaps by using XDoclet in the build process. Then, in the model constructor (which runs prior to the parameter panel being constructed), the XML descriptors can be read in, and a pull-down list added to the initial model parameters. Given the chosen class, in the model `begin()` method, we can dynamically load the correct factory class and call its `generatePopulation()` method.

NOTE: As of 3/1/2007, I'm still working on this particular architecture. At the moment the correct `IPopulationFactory` is referenced statically in the model class, and the population types are hardcoded into the model constructor. This is very undesirable for clean expansion of the codebase, but I'm in transition as of the current version.

4.7. Cleaning Up Model Classes

In creating a modular simulation template, it is turning out that the main model class is both turning largely into boilerplate code and gaining some critical internal complexity to handle the generic nature of the other simulation structures. At the same time, the model class itself does represent a key piece of code for the modeler—containing simulation-level parameters, data structures, and so on. The modeler needs to construct a schedule for the simulation, and instantiate.

Solution here is to probably subclass all my generic mechanisms from `SimModelImpl` and have our models subclass from there, supplying a schedule, parameters, and any other model-level data fields.

4.8. Miscellaneous Notes

Several problems still exist that I need to work on, in addition to the generalization work itself.

1. The fixed-size arrays that remain in the model class are somewhat fragile depending upon the combination of maxVariants and numNodes, now that the population is constructed by the new factory pattern, which knows nothing about "maxVariants." I've tried to work on this by having `IAgentPopulation` classes track

the "largest" variant they create during construction, and making that available to models via `getCurrentMaximumVariant()`. Then, in the `mutateVariants()` method, we increment the current notion of the "biggest" variant. This is then used to re-allocate the various "top40" arrays in the original model methods. The unfortunate thing is that I have a bug on the first time step if numNodes > initial MaxVariants. At the moment the work-around is to ensure that the initial MaxVariants value is high enough, but I'll fix this bug soon.

Acknowledgements

Thanks to Alex Bentley for providing the code from their 2007 paper and agreeing to collaboration on a new generation of the model, and allowing free distribution for others to use as well.