The Art of Iterating: Update-Strategies in Agent-Based Simulations

Jonathan Thaler

January 20, 2017

Abstract

When developing a model for an Agent-Based Simulation (ABS) it is of very importance to select the right update-strategy for the agents to produce the desired results. In this paper we develop a systematic treatment of all general update-strategies in ABS and discuss their philosophical interpretation and meaning. Further we discuss the suitability of the very three different programming languages Java, Scala with Actors and Haskell to implement each of the update-strategies. Thus this papers contribution is the development of a general terminology of update-strategies and their implementation issues in various kinds of programming languages.

1 Introduction

In the paper of [4] the authors showed that the results of the simulation of the classic prisoners-dilemma on a 2D-grid reported in [5] depends on a a very specific strategy of iterating this simulation and show that the beautiful patterns as reported by [5] will not form when selecting a different iteration-strategy. Although the authors discussed philosophical aspects of choosing one strategy over the other, they lacked to generalize their observation. We will do so in the central message of our paper by stressing that when doing Agent-Based Simulation & Modelling (ABM/S) *it is of most importance to select the right iteration-strategy which reflects and supports the corresponding semantics of the model*. We find that this awareness is yet still under-represented in the literature of ABM/S and most important of all is lacking a systematic treatment. Thus our contribution in this paper is to provide such a systematic treatment by

- Presenting all the general iteration-strategies which are possible in an ABM/S.
- Developing a systematic terminology of talking about them.
- Giving the philosophical interpretation and meaning of each of them.

- Comparing the 3 programming languages Java, Haskell and Scala in regard of their suitability to implement each of these strategies.

Besides the systematic treatment of all the general iteration-strategies the paper presents another novelty which is its inclusion of the pure functional declarative language Haskell in the comparison. This language has so far been neglected by the ABM/S community which is dominated by object-oriented (OO) programming languages like Java thus the usage of Haskell presents a real, original novelty in this paper.

2 Problem

1 Page

- describe in more technical detail what the introduction tells.
- [4] and `https://www.openabm.org/book/33102/54-importance-sequence-updating` only mentions synchronous and asynchronous updates but this is not precise enough and lacks subcategories

question: do emergent patterns break down / global dynamics change completely in some ABM/S when changing sim-semantic? which kind of ABM could show this behaviour? which properties are responsible for it? Answer: Yes they do but only under given circumstances: discrete simulations with dependence on each other. continuous not so easy.
do we find a continuous simulation in which it breaks down under given circumstances? Yes: Heroes & Cowards can lead to specific patterns as shown by the creators

2.1 Simulations

Today simulations are at the very heart of many sciences. They allow to put hypotheses to test by building a model which abstracts from reality, keeping only the important and relevant details, and then bringing this model to life in simulation. Based on the results shown by the dynamics, previously formulated hypotheses can be verified or falsified resulting in a formulate-simulate-refine cycle.
The meaning of simulating a model can be understood as calculating the dynamics of a (model of a) system over time thus the state of the system at time t depends on the state of the system at time t - epsilon. Here we only consider simulations in a computer-system (TODO: are there simulations NOT in a computer?), which is an inherently discrete system which poses us with the question of how to represent time which seems linear and continuously flowing to us in reality (NOTE: this may not be physically the case but for our considerations this should be a good approximation). Being in a discrete system, of course implies that time has to be discretised as well and there are two ways of doing it: discrete and continuous where in discrete case time advances in steps of

the natural numbers and where in the continuous case time advances in steps of real-numbers. Note that in both cases the system is iterated in steps where only the *numerical type* of the input to the time-dependent functions differs. Thus a simulation in a computer can be understood as an iteration over a model for a given number of steps where each step advances time by dt (either discrete or continuous) and, based on the previous model-state, producing an updated model-state which again becomes the input for the next step. Thus in each simulation we have three inputs: 1. the model, 2. number of steps, 3. time dt. There are of course different models and types of simulations and in this paper we will focus on one particular one: agent-based, which will be described next.

2.2 Agent-Based Modelling and Simulation (ABM/S)

ABM/S is a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is of knowledge [6]. Those parts, called Agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. Thus the central aspect of ABM/S is the concept of an Agent which can be understood as a metaphor for a pro-active unit, able to spawn new Agents, and interacting with other Agents in a network of neighbours by exchange of messages. The implementation of Agents can vary and strongly depends on the programming language and the kind of domain the simulation and model is situated in. Whereas the majority of ABM/S are implemented in object-oriented (OO) languages e.g. Java, C++, this paper focuses on functional ones.

3 Update-Strategies

2 Pages
This is all programming-language agnostic

- A terminology and classification of all the possible iteration-strategies presented as a list
- short discussion separate paragraph for each
  - Abstract implementation of the strategies
  - Philosophical meaning and interpretation
  - Advice for selecting it

3.1 Classification

Name	*Time*	*Order*	*Decisions*	*Non-Deterministic*	*Type*
Sequential	Global	Sequential	Global	No	Sync
Parallel	Global	Parallel	Local	No	Sync
Concurrent	Global	Concurrent	Global	Yes	Async
Actors	Local	Random	Local	Yes	Async

Table 1: Summary of simulation-stepping methods.

3.2 Sequential - Strategy

TODO: deterministic iteration, random-iteration with uniform distribution TODO: keep time constant for each agent in one iteration OR advance for every agent by fraction of dt: agent-time = t + (ai * dt/n) where t is the current simulation time, ai the agents index, dt the amount of time the simulation will be advanced by and n the number of agents. In the end the new current time will be then tnext = tcurr + dt other possibilities of advancing is agent-time = t + ai * dt. in the end the new current time will be then tnext = tcurr + n * dt

update one Agent after another. We assume that, given the updates are done in order of the index $i1ton$, then Agents $a_{n>i}$ see the updated agent-state / influence on the environment of agent a_i. Note that if this is not the case we would end up in the parallel-case (see next) *independent* whether it is in fact running in parallel or not. For breaking deterministic ordering which could result in giving an Agent an advantage (e.g. having more information towards the end of the step) one could implement a random-walk in each step but this does not fundamentally change this approach. Also if one thinks the simulation continuously, where each step is just a very small update like in Heroes & Cowards, then the random ordering should not change anything fundamental as no agent has real information-benefit over others as there is continuous iteration thus the agent once ahead is then behind. TODO: maybe need to make more formal

3.3 Parallel - Strategy

update all Agents in parallel. This case is obviously only possible if the agents cannot interfere with each other or the environment through shared state. In this case it will make no difference how we iterate over the agents, the outcome *has to be* the same - it is event-ordering invariant as all events/updates happen *virtually* at the *same time*. Haskell is a strong proponent of this implementation-technique.

If one wants to write global in case of parallel this is regarded as a systematic error as this is not logical as it would imply an ordering thus we requiring different semantics: SEQ or CONCURRENT. Thus we would have to make the Environment in case of Par local to an agent which is the same as moving it into the agents state =¿ we choose another approach: pass in an environment which cant be changed by the agents (no environment in return type) but only by the simulation-iterator after an iteration. =¿ dynamic WildFire-Model does not work with PAR

3.4 Concurrent - Strategy

update all Agents concurrently. In this case the agents run in parallel but share some state which access has to be synchronized thus introducing real random event-orderings which may or may not be desirable in the given simulation model. Can be implemented in both Java and Haskell.

3.5 Actors - Strategy

TODO: discuss how local-time can be handled: real-time or simulation-time - its always local and not synchronized globally because then we would end up in Concurrent Strategy
in the Act-version we need to observe the agents: we need to sample them regularly =¿ we have all the issues with sampling
In this case there is no global iteration over steps but all the Agents run in parallel, doing local stepping and communicate with each other either through shared state or messages. Note that this does not impose any specific ordering of the update and can thus regarded to be real random due to its concurrent nature. It is possible to simulate the global-stepping methods from above by introducing some global locking forcing the agents into lock-step. This is the approach chosen for Scala & Actors.

3.6 Update-Strategies

1. All states are copied/frozen which has the effect that all agents update their positions *simultaneously*

2. Updating one agent after another utilizing aliasing (sharing of references) to allow agents updated *after* agents before to see the agents updated before them. Here we have also two strategies: deterministic- and random-traversal.

3. Local observations: Akka

3.7 Different results with different Update-Strategies?

Problem: the following properties have to be the same to reproduce the same results in different implementations:

Same initial data: Random-Number-Generators Same numerical-computation: floating-point arithmetic Same ordering of events: update-strategy, traversal, parallelism, concurrency

- Same Random-Number Generator (RNG) algorithm which must produce the same sequence given the same initial seed.
- Same Floating-Point arithmetic
- Same ordering of events: in Scala & Actors this is impossible to achieve because actors run in parallel thus relying on os-specific non-deterministic scheduling. Note that although the scheduling algorithm is of course deterministic in all os (i guess) the time when a thread is scheduled depends on the current state of the system which can change all the time due to *very* high number of variables outside of influence (some of the non-deterministic): user-input, network-input, ... which in effect make the system appear as non-deterministic due to highly complex dependencies and feedback.
- Same dt sequence =¿ dt MUST NOT come from GUI/rendering-loop because gui/rendering is, as all parallelism/concurency subject to performance variations depending on scheduling and load of OS.

It is possible to compare the influences of update-strategies in the Java implementation by running two exact simulations (agentcount, speed, dt, herodistribution, random-seed, world-type) in lock-step and comparing the positions of the agent-pairs with same ids after each iteration. If either the x or y coordinate is no equal then the positions are defined to be *not* equal and thus we assume the simulations have then diverged from each other.
It is clear that we cannot compare two floating-point numbers by trivial == operator as floating-point numbers always suffer rounding errors thus introducing imprecision. What may seem to be a straight-forward solution would be to introduce some epsilon, measuring the absolute error: abs(x1 - x2) ¿ epsilon, but this still has its pitfalls. The problem with this is that, when number being compared are very small as well then epsilon could be far too big thus returning to be true despite the small numbers are compared to each other quite different. Also if the numbers are very large the epsilon could end up being smaller than the smallest rounding error, so that this comparison will always return false. The solution would be to look at the *relative error*: abs((a-b)/b) ¡ epsilon.
The problem of introducing a relative error is that in our case although the relative error can be very small the comparison could be determined to be different but looking in fact exactly the same without being able to be distinguished with the eye. Thus we make use of the fact that our coordinates are virtual ones, always being in the range of [0..1] and are falling back to the measure of absolute error with an epsilon of 0.1. Why this big epsilon? Because this will then definitely show us that the simulation is *different*.

The question is then which update-strategies lead to diverging results. The hypothesis is that when doing simultaneous updates it should make no difference when doing random-traversal or deterministic traversal =¿ when comparing two simulations with simultaneous updates and all the same except first random-

and the other deterministic traversal then they should never diverge. Why? Because in the simultaneous updates there is no ordering introduce, all states are frozen and thus the ordering of the updates should have no influence, *both simulations should never diverge,* ***independent how dt and epsilon are selected***.
Do the simulation-results support the hypothesis? Yes they support the hypothesis - even in the worst cast with very large dt compared to epsilon (e.g. dt = 1.0, epsilon = 1.0-12)
The 2nd hypothesis is then of course that when doing consecutive updates the simulations will *always* diverge independent when having different traversal-strategies.
Simulations show that the selection of *dt* is crucial in how fast the simulations diverge when using different traversal-strategies. The observation is that *The larger dt the faster they diverge and the more substantial and earlier the divergence.*. Of course it is not possible to proof using simulations alone that they will always diverge when having different traversal-strategies. Maybe looking at the dynamics of the error (the maximum of the difference of the x and y pairs) would reveal some insight?

The 3rd hypothesis is that the number of agents should also lead to increased speed of divergence when having different traversal-strategies. This could be shown when going from 60 agents with a dt of 0.01 which never exceeded a global error of 0.02 to 6000 agents which after 3239 steps exceeded the absolute error of 0.1.

3.8 Reproducing Results in different Implementations

actors: time is always local and thus information as well. if we fall back to a global time like system time we would also fall back to real-time. anyway in distributed systems clock sync is a very non-trivial problem and inherently not possible (really?). thus using some global clock on a metalevel above/outside the simulation will only buy us more problems than it would solve us. real-time does not help either as it is never hard real time and thus also unpredictable: if one tells the actor to send itself a message after 100ms then one relies on the capability of the OS-timer and scheduler to schedule exactly after 100ms: something which is always possible thus 100ms are never hard 100ms but soft with variations.
qualitative comparison: print pucture with patterns. all implementations are able to reproduce these patterns independent from the update strategy
no need to compare individual runs and waste time in implementing RNGs, what is more interesting is whether the qualitative results are the same: does the system show the same emergent behaviour? Of course if we can show that the system will behave exactly the same then it will also exhibit the same emergent behaviour but that is not possible under some circumstances e.g. the simulation-runs of Akka are always unique and never comparable due to random event-ordering produced by concurrency & scheduling. Also we don't have to

proof the obvious: given the same algorithm, the same random-data, the same treatment of numbers and the same ordering of events, the outcome *must* be the same, otherwise there are bugs in the program. Thus when comparing results given all the above mentioned properties are the same one in effect tests only if the programs contain no bugs - or the same bugs, if they *are the same*.

Thus we can say: the systems behave qualitatively the same under different event-orderings.
Thus the essence of this boils down to the question: "Is the emergent behaviour of the system is stable under random/different/varying event-ordering?". In this case it seems to be so as proofed by the Akka implementation. In fact this is a very desirable property of a system showing emergent behaviour but we need to get much more precise here: what is an event? what is an emergent behaviour of a system? what is random-ordering of events? (Note: obviously we are speaking about repeated runs of a system where the initial conditions may be the same but due to implementation details like concurrency we get a different event-ordering in each simulation-run, thus the event-orderings vary between runs, they can be in fact be regarded as random).

4 Implementation Approaches

5 Pages
This is now very programming-language specific

- Mapping the strategies to 3 programming-languages: Java, Scala with Actors, Haskell
- Comparing the programming languages in regard of their suitability to implement each of these strategies
- Screen-shots of results of the same simulation-model with all the strategies

4.1 Selection of the Languages

-¿ Java: supports global data =¿ suitable to implement global decisions: implementing global-time, sequential iteration with global decisions -¿ Haskell: has no global data =¿ local decisions (has support for global data through STM/IO but then looses very power?) =¿ implementing global-time, parallel iteration with local-decisions. -¿ Haskell STM solution =¿ implementing concurrent version using STM? but this is very complicated in its own right but utilizing STM it will be much more easier than in java -¿ Scala: mixed, can do both =¿ implementing local time with random iteration and local decisions

4.2 Java

sequential more natural in java, parallel needs to "think functional" in java concurrency and actors always difficult in java despite java provides very good synchronization primitives

4.3 Scala with Actors

direct support for actors

4.4 Haskell[1]

note that the difference between SEQ and PAR in Haskell is in the end a 'fold' over the agents in the case of SEQ and a 'map' in the case of PAR dont have objects with methods which can call between each other but we need some way of representing agents. this is done using a struct type with a behaviour function and messaging mechanisms. important: agents are not carried arround but messages are sent to a receiver identified by an id. This is also a big difference to java where don't really need this kind of abstraction due to the use of objects and their 'messaging'. messaging mechanisms have up- and downsides, elaborate on it.
concurrency and actors extremely elegant possible through: STM which only possible in languages without side-effect
the conc-version and the act-version of the agent-implementations look EXACTLY the same BUT we lost the ability to step the simulation!!!
This is the process of implementing the behaviour of the Agent as specified in the model. Although there are various kinds of Agent-Models like BDI but the basic principle is always the same: sense the environment, process messages, execute actions: change environment, send messages. According to [6] and also influenced by Actors from [2] one can abstract the abilities in each step of an Agent to be the following:

1. Process received messages
2. Create new Agents
3. Send messages to other Agents
4. Sense (read) the environment
5. Influence (write) the environment

5 Related Research

- [4]
- [1]

[1] Code available under
`https://github.com/thalerjonathan/phd/tree/master/coding/papers/iteratingABM/haskell`

[3] sketch a minimal agent-framework in Haskell which is very similar in the basic structure of ours, also utilizing an agent-transforming function which consumes incoming messages and produces outgoing ones. This proofs that this approach, very well developed in ABM/S, seems to be a very natural one also to apply to Haskell. Their focus is more on economic simulations and instead of iterating a simulation with a global time, their focus is on how to synchronize agents which have internal, local transition times. They introduce a time-keeper agent which synchronizes the actions of all of the agents thus we argue that our framework is able to capture it faithfully using the *Actor-Strategy* utilizing either a timer-keeper as they do or through the access of the global shared-environment. Although their work uses Haskell as well, this does not diminish the novelty of our approach using Haskell because our focus is very different from them.

6 Conclusion

- Selecting the correct Iteration-Strategy is of most importance and must match the model semantics
- Java: best for non-parallel, non-concurrent strategy
- Scala with Actors: best for concurrency
- Surprise: Haskell can faithfully implement all strategies equally well, something not anticipated in the beginning

6.1 Haskell excells

We initially thought that Haskell would be suitable best for just implementing the Par-Strategy after implementing all the strategies in it we found out that Haskell is extremely well suited to implement all the strategies.
We think this stems from the following facts: no side-effects (unless reflected in the types): is a must-have for STM, although it makes things more difficult in the beginning, in the end it turns out to be a blessing because one can guarantee that side-effects won't occur. We have taken care that the agents all run in side-effect free code.
STM: implementing concurrency is a piece-of-cake
extremely powerful static typesystem: in combination with side-effect free this results in the semantics of an update-strategy to be reflected in the Agent-Transformer function and the messaging-interface. This means a user of this approach can be guided by the types and can't abuse them. Thus the lesson learned here is that *if one tries to abuse the types of the agent-transformer or work around, then this is an indication that the update-strategy one has selected does not match the semantics of the model one wants to implement.* If this happens in Java, it is much more easier to work around by introducing global variables or side-effects but this is not possible in Haskell.
Thus our conclusion in using Haskell is that it is an extremely underestimated

language in ABM/S which should be explored much more as we have shown that it really shines in this context and we believe that it could be pushed further even more.

7 Further Research

- Yampa
- Reasoning in Haskell about the Model & Simulation
- Develop a small modelling-language which is close to the Haskell-Version of modelling agents therefore specification and implementation match

7.1 Functional Reactive Programming

The implemented framework in Haskell is lacking features like TODO and is basically an attempt of reinventing Functional Reactive Programming (FRP). We were aware of the existence of this paradigm, especially the library Yampa, but decided to leave that one to a side and really keep our implementation clear and very basic. The next step would be to fusion our implementations with Yampa thus leveraging both approaches.

References

[1] A, M. L., B, R. D., and B, K. R. *A Framework for Megascale Agent Based Model Simulations on the GPU.* 2008.

[2] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[3] Botta, N., Mandel, A., and Ionescu, C. Time in discrete agent-based models of socio-economic systems. Documents de travail du Centre d'Economie de la Sorbonne 10076, Université Panthéon-Sorbonne (Paris 1), Centre d'Economie de la Sorbonne, 2010.

[4] Huberman, B. A., and Glance, N. S. Evolutionary games and computer simulations. *Proceedings of the National Academy of Sciences 90*, 16 (Aug. 1993), 7716–7718.

[5] Nowak, M. A., and May, R. M. Evolutionary games and spatial chaos. *Nature 359*, 6398 (Oct. 1992), 826–829.

[6] Wooldridge, M. *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, 2009.