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

Jonathan Thaler

January 24, 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 [8] the authors showed that the results of the simulation of the classic prisoners-dilemma on a 2D-grid reported in [9] depends on a a very specific strategy of iterating this simulation and show that the beautiful patterns as reported by [9] will not form when selecting a different iteration-strategy. Although the authors differentiated between two strategies, their description still lacks precision and generality which we will try to repair in this paper. Although they too 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 Related Research

TODO: [8] where the first to discuss the differences update-strategies can make
TODO: [1] give an approach for ABM/S on GPUs. a seemingly completely different form but which, we hypothesize, can be roughly mapped to our PAR-Strategy.
[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.
TODO: [11]: Master-Thesis "The Agent-based Simulation Environment in Java" by Yuxuan Jin under supervision of Peer-Olaf
TODO: [5]: comparison of c++ and java for ABM/S

3 The Problem

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 [10]. 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 unique pro-active unit, able to spawn new Agents, interacting with other Agents in a network of neighbours by exchange of messages which are situated in a generic environment. Thus we informally assume the following about our Agents:

- They have a unique identifier
- They can initiate actions on their own e.g. change their own state, send messages, create new agents, kill themselves,...
- They can react to messages they receive with actions (see above)
- They can interact with a generic environment they are situated in

An implementation of an ABS must solve thus solve three fundamental problems:

1. Source of pro-activity
   How can an Agent initiate actions without external stimuli?
2. Message-Processing
   When is a message m, visible to Agent B, processed by it?
3. Semantics of Message-Delivery
   When is a message m, sent by Agent A to Agent B, visible to B?

In computer systems, pro-activity, the ability to initiate actions on its own without external stimuli, is only possible when there is some internal stimuli, most naturally represented by some generic notion of monotonic increasing time-flow. This can either be some physical real-time system-clock which counts the milliseconds since 1970 (thus binding the time-flow of the system to the one of the 'real-world') or a virtual simulation-clock which is just a monotonic increasing natural number. As we are in a discrete computer-system, this time-flow must be discretized as well in discrete steps and each step must be made available to the Agent, acting as one internal stimuli. This allows the Agent then to perceive time and become pro-active depending on time (NOTE: we could argue that this is not really pro-activity because it depends always on time, but there is really now other way of doing this in our current implementation of computer-systems.). Independent of the representation of the time-flow we have the two fundamental choices whether the time-flow is local to the Agent or whether it is a system-global time-flow.

The semantics of message-delivery define when sent messages are visible to the receivers so they can process them and react to them. The only two ways of implementing them are that messages are visible either *immediately* or after a synchronization point between the sender and receiver. Such a synchronization point can be a local one, just between the two or a global one between all Agents in the system.

Basically we can say that we want to process a message as soon as it is visible to us but this is not how real computer-systems can work. In a real system each Agent would have a message-box into which the messages are posted so the Agent can then check its mail-box for new messages. The question is then when the Agent is going to poll for new messages? Clearly what we need is

a recurring, regular trigger which allows the Agent to poll the mail-box and process all queued messages. We argue that the most natural approach is to bind this trigger to the time-flow step which provides pro-activity.

To solve these problems an update-strategy is implemented which will iterate through the Agents in *some* way and allow the Agents to perform these steps. It is immediately clear that different choices in the specific problems will lead to different system behaviour. To discuss this we will first present all possible update-strategies and their details in the next section and then outline how they influence the system behaviour.

4 Update-Strategies

In this section we will present all the update-strategies which are available in ABM/S in a general form, discuss abstract details independent from programming languages, give philosophical meaning and interpretation of them and advices for selecting them for which model.

Name	*Time-Flow*	*Iteration-Order*	*Deterministic*
Seq	Global	Sequential	Yes
Par	Global	Parallel	Yes
Con	Global	Concurrent	No
Act	Local	Actor	No

Table 1: List of all general update-strategies in ABM/S.

4.1 Seq

Description: This strategy has a global time-flow and in each time-step iterates through all the agents and updates one Agent after another. Messages sent and changes to the environment made by Agents are visible immediately. More formally, we assume that, given the updates are done in order of the index $i = [1..n]$, then Agents $a_{n>i}$ see the changes to environment and messages sent to them by Agent a_i. Note that there is no source of randomness and non-determinism thus rendering this strategy to be completely deterministic in each step.

Time-Updates: The most straight-forward approach is to keep the time constant for each agent in one iteration. This though may seem non-logic as the actions of preceding Agents are visible to later ones but time has not changed since then. Thus if the model semantics require that time changes with every Agent we could 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

Extensions: If the sequential iteration from 1..n imposes an advantage over the Agents further ahead or behind in the queue (e.g. if it is of benefit when making choices earlier than others in auctions or later when more information is available) then one could use random-walk iteration where in each time-step the agents are shuffled before iterated. Note that although this would introduce randomness in the model the source is a random-number generator thus reproduce-able.

4.2 Par

Description: This strategy has a global time-flow and in each time-step iterates through all the agents and updates all Agents in parallel. Messages sent and changes to the environment made by Agents are visible in the next global step. We can think about this strategy that all Agents make their moves at the same time.
Environment: If one wants to change the environment in a way that it would be visible to other Agents this is regarded as a systematic error in this strategy. First it is not logical because all actions are meant to happen at the same time and also it would implicitly induce an ordering thus violating the *happens at the same time* idea. Thus we require different semantics for accessing the environment in this strategy. We introduce thus a *global* environment which is made up of the set of *local* environments. Each local environment is owned by an Agent thus there are as many local environments as there are Agents. The semantics are then as follows: in each step all Agents can *read* the global environment and *read/write* their local environment. The changes to a local environment are only visible *after* the local step and can be fed back into the global environment after the parallel processing of the Agents.
Time-Updates: Time really stays constant in this case for all Agents in one step as all updates happen really at the same *virtual* time. It would make no sense to advance the time between Agents as all actions are meant to happen at the same time, without imposing any ordering amongst them.
Semantics: It does not make a difference if the Agents are really computed in parallel or just sequentially, due to the semantics of changes, this has the same effect. In this case it will make no difference how we iterate over the agents (sequentially, randomly), the outcome *has to be* the same - it is event-ordering invariant as all events/updates happen *virtually* at the *same time*. Thus if one needs to have the semantics of writes on the whole (global) environment in ones model, then this strategy is not the right one and one should resort to one of the other strategies.

4.3 Con

Description: This strategy has a global time-flow and in each time-step iterates through all the agents and updates all Agents in parallel but all messages sent and changes to the environment are immediately visible. Thus this strategy can

be understood as a mix of Seq and Par: all Agents run at the same time with actions becoming immediately visible.
Semantics: It is important to realize that, when running Agents in parallel which are able to see actions by others immediately, this is the very definition of concurrency: parallel execution with mutual read/write access to shared data. Of course this shared data-access needs to be synchronized which in turn will introduce event-orderings in the execution of the Agents. Thus at this point we have a source of inherent non-determinism: although we would ignore any hardware-model of concurrency at some point we need arbitration to decide which Agent gets access first to a shared resource thus arriving at non-deterministic solutions - this will become much clearer in the results-section. This has the very important influence that repeated runs with the same configuration of the Agents and the Model may lead to different results each time.

4.4 Act

Description: This strategy has no global time-flow but all the Agents run concurrently in parallel, with their own local time-flow. The messages and changes to the environment are visible as soon as the data arrive at the local Agents - this can be immediately when running locally on a multi-processor or with a significant delay when running in a cluster over a network. Obviously this is also a non-deterministic strategy and repeated runs with the same Agent and Model-configuration may (and will) lead to different results.
Locality of Information: It is of most importance to note that information and thus also time in this strategy is always local to an Agent as each Agent progresses in its own speed through the simulation. Thus in this case one needs to explicitly *observe* an Agent when one wants to e.g. visualize it. This observation is then only valid for this current point in time, local to the observer but not to the Agent itself, which may have changed immediately after the observation. This implies that we need to sample our Agents with observations when wanting to visualize them, which would inherently lead to well known sampling issues. A solution would be to invert the problem and create an Observer-Agent which is known to all Agents where each Agent sends a *'I have changed'* message with the necessary information to the observer if it has changed its internal state. This also does not guarantee that the observations will really reflect the actual state the Agent is in but is a remedy against the notorious sampling.
Semantics: This is the most general one of all the strategies as it can emulate all the others by introducing the necessary synchronization mechanisms and Agents. Also this concept was proposed by C. Hewitt in 1973 in his work [7] where upon I. Grief in [6] and W. Clinger in [4] developed semantics of different kinds. These works were very influential in the development of the Agent-Term and concept and can be regarded as foundational basics for ABM/S.

5 Results

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

5.1 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
Note that we didn't select a language where the memory-management falls in the hands of the developer. This would be the case e.g. in C++. This was looked into partially by [5] but the focus of this paper is not on this issue as it would complicated things dramatically. All used languages are garbage-collected / the developer does not need to care how memory is cleared up.

5.2 Selected Models

5.2.1 Heroes & Cowards

5.2.2 SIRS

5.2.3 Wildfire

5.2.4 Spatial Game

5.3 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
IMPORTANT: no need to explicitly add the environment, as this can be individually implemented by the agents on a shared basis (references)
sequential is able to work completely without messages and only by accesing references to neighbours but we explicitly don't want to follow this obvious

way and stick to the send/receive message paradigm. we keep Agent-instance references
parallel then needs to utilize messages because would violate the parallel implementation. TODO: split state from agent and only update agent-state problem: can send references through messages: share data although the interfaces encourage it we cannot prevent the agents to use agent-references and directly accessing. a workaround would be to create new agent-instances in every iteration-step which would make old references useless but this doesn't protect us from concurrency issues with a current iteration (the copying must take place within a synchronized block, thus implicitly assuming ordering, something we don't want) and besides, can always work around and update the references. thats the toll of side-effects: faster execution but less control over abuse tried to clone agents in each step and let them collect their messages =¿ extremely slow
conc: expect it to be a pain in the ass with java. it is not: its the same interface as in SEQ with updates running parallel like in PAR but
act: need to copy messages, otherwise could stuck in an endless loop

5.4 Haskell[1]

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.
but still it is not suitable for big models with heterogeneous agents, there things are lacking: see further research

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

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 [10] 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

5.5 Scala with Actors

direct support for actors

6 Conclusion

- Selecting the correct Iteration-Strategy is of most importance and must match the model semantics
- Java: parallelism and concurrency very easy due to elegant and powerful built-in synchronization primitives, high-performance and large number of agents possible due to aliasing and side-effects
- Surprise: Haskell can faithfully implement all strategies equally well, something not anticipated in the beginning. But: still much work to do to capture large and complex models (see further research), performance is a big issue but this has not been about performance (2000 Agents are enough)
- Scala with Actors: extremely elegant solutions possible

7 Further Research

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 from which we hypothesize to gain the ability to develop much more complex models with heterogeneous agents.

7.2 Develop a small modelling-language which is close to the Haskell-Version of modelling agents therefore specification and implementation match

See paper

7.3 Reasoning in Haskell about the Model & Simulation

TODO: sketch ideas

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] Clinger, W. D. Foundations of Actor Semantics. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[5] Dawson, D., Siebers, P. O., and Vu, T. M. Opening pandora's box: Some insight into the inner workings of an Agent-Based Simulation environment. In *2014 Federated Conference on Computer Science and Information Systems* (Sept. 2014), pp. 1453–1460.

[6] Grief, I., and Greif, I. SEMANTICS OF COMMUNICATING PARALLEL PROCESSES. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[7] Hewitt, C., Bishop, P., and Steiger, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 1973), IJCAI'73, Morgan Kaufmann Publishers Inc., pp. 235–245.

[8] 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.

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

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

[11] Yuxuan, J. *The Agent-based Simulation Environment in Java.* PhD thesis, University Of Nottingham, School Of Computer Science, 2016.