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

Jonathan Thaler

February 2, 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 properties, derive the possible update-strategies in ABS and discuss their interpretation and semantics something which is still lacking in the literature on ABS. Further we investigate the suitability of three very different programming languages Java, Haskell and Scala with Actors to implement each of the update-strategies. Thus this papers contribution is the development of a new, general terminology of update-strategies and their implementation comparison in various kinds of programming languages.

Keywords

Agent Based Simulation, Simulation, Parallelism, Concurrency, Haskell, Actors

1 Introduction

In the paper [11] the authors showed that the results of simulating the classic prisoners-dilemma game on a 2D-grid as reported in [14] depends on a a very specific strategy of iterating this simulation and show that the beautiful patterns seen in figure 1 will not form when selecting a different iteration-strategy.

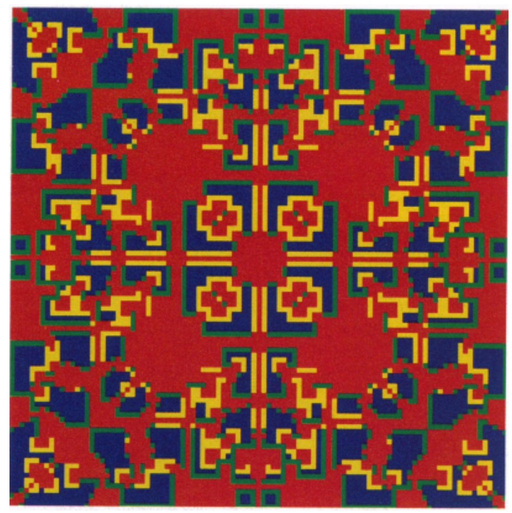


Figure 1: Patterns formed by playing the prisoners-dilemma game on a 2D-grid using a *synchronous* update-strategy. Picture taken from [11].

Although the authors differentiated between two strategies, their description still lacks precision which we will try to give in this paper. They also discussed philosophical aspects of choosing one strategy over the other, but lacked to generalize their observation. We will do so in the central message of our paper by stressing that when doing ABS *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 ABS and lacking a systematic treatment. Thus our contribution in this paper is to provide a such by

- Presenting general properties of ABS and deriving update-strategies.
- Developing a new, general terminology of talking about the update-strategies.
- Giving the semantic interpretation and meaning of each of them.
- Comparing the three programming languages Java, Haskell and Scala with Actors in regard of their suitability to implement each of these strategies.

It is important to note that the amount of research of using Haskell in the field of ABS has so far been moderate. Though there exist a few papers which look into Haskell and ABS [6], [18], [13] all treat Haskell in this context very generally and focus primarily on how to specify Agents. This papers is looking at fundamental technical details of Haskell's suitability in implementing update-strategies in ABS, something not looked at in the ABS community thus presenting an original novelty.
TODO [17]

2 Related Research

Already noted in the introduction, [11] where the first to discuss the differences update-strategies can make and introduced the terms of synchronous and asynchronous updates. They define to be synchronous as Agents being updated in unison and asynchronous where one Agent is updated and the others are held constant.

[1] give an approach for ABS on GPUs which is a very different approach to updating and iterating Agents in ABS. They discuss execution order at length, highlight the problem of inducing a specific execution-order in a model which is problematic for parallel execution and give solutions how to circumvent these shortcomings. Although we havn't mapped our ideas to GPUs we explicitly include an approach for data-parallelism which, we hypothesize, can be utilized to roughly mapped their approach onto our terminology.

[3] sketch a minimal agent implementation in Haskell which is very similar in the basic structure of ours. This proves that our approach seems to be a very natural one also to apply to Haskell. Their focus is primarily 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. Although their work uses Haskell as well, this does not diminish the novelty of our approach using Haskell as our focus is very different from theirs and approaches ABS in a more general and comprehensive way.

[5] describe basic inner workings of ABS environments and compare their implementation in C++ to the existing ABS environment AnyLogic which is programmed in Java. They explicitly mention asynchronous and synchronous time-models and compare them in theory but unfortunately couldn't report the results of asynchronous updates due to limited space. They interpret asynchronous time-models to be the ones in which an Agent acts at random time intervals and synchronous time-models where Agents are updated all in same time intervals.

[22] presents in his Master-Thesis a comprehensive discussion on how to implement an ABS for state-charts in Java and also mentions synchronous and asynchronous time-models. He identifies the asynchronous time-model to be one in which updates are triggered by the exchange of messages and the synchronous ones which trigger changes immediately without the indirection of messages.

We observe that there seems to be a variety of meanings attributed to the terminology of asynchronous and synchronous updates but the very semantic and technical details are unclear and not described very precisely. In the next we will address this issue by presenting the basic background and propose properties for a new terminology from which we can drive common update-strategies.

3 Background

3.1 Agent-Based Simulation

ABS 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. 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 ABS is the concept of an Agent which can be understood as a metaphor for a pro-active unit, situated in a generic environment, able to spawn new Agents and interacting with other Agents in a network of neighbours by exchange of messages [21]. Thus we informally assume the following about our Agents:

- They have a unique identifier and some internal state
- They can initiate actions on their own e.g. change their internal 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 thus solve two fundamental problems:

1. **Source of pro-activity** How can an Agent initiate actions without the external stimuli of messages?
2. **Semantics of Messaging** When is a message m, sent by Agent A to Agent B, visible and processed by 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. Due to the discrete nature of computer-system, this time-flow must be discretized in steps as well and each step must be made available to the Agent, acting as the internal stimuli. This allows the Agent then to perceive time and become pro-active depending on time. Thus we can understand an ABS as a discrete time-simulation where time is broken down into continuous, real-valued or discrete natural-valued time-steps. 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. Time-flows in computer-systems can only be created through threads of execution thus there are two of ways of feeding time-flow into an Agent. Either it has its own thread-of-execution or the system creates the illusions of their own thread-of-execution by sharing the global one sequentially among the Agents where an Agent has to yield the execution back after it has executed its step. Note the parallels to an operating system with cooperative multitasking in the latter case and real multiprocessing in the former.

The semantics of messaging define when sent messages are visible to the receivers and when the receivers process them. Message-processing could happen either immediately or delayed, depending on how message-delivery works. There are two ways of message-delivery: queued or immediate. In the case of immediate message-deliver the message is sent directly to the Agent without any queuing in between e.g. a direct method-call. This would allow an Agent to immediately react to this message as this call of the method transfers the thread-of-execution to the Agent. This is not the case in the queued message-delivery where messages are posted to the message-box of an Agent and the Agent pro-actively processes the message-box at regular points in time.

3.2 ABS Properties

To develop a new terminology, we propose to abandon the notion of synchronous and asynchronous updates and, based on the discussion above we propose six properties characterizing the dimensions and details of the internals of an ABS:

Iteration-Order Is the collection of Agents updated *sequential* with one Agent updated after the other or are all Agents updated in *parallel*, at virtually the same time?

Global Synchronization Is a full Iteration over the collection of Agents happening in lock-step at global points in time or not (*yes/no*)?

Thread of Execution Does each Agent has a *separate* thread-of-execution or does it *share* it with all the others? Note that it seems to have a constraint on the Iteration-Order, namely that *parallel* execution forces separate threads of execution for all Agents. We will show that this is not the case, when looking at the Parallel Strategy in the next section.

Message-Handling Are messages handled *immediately* by an Agent when sent to them or are they *queued* and processed later? Here we have the constraint, that an immediate reaction to messages is only possible when the Agents share a common thread of execution. Note that we must enforce this constraint as otherwise Agents could end up having more than one thread-of-execution which could result in them acting concurrently by making simultaneous actions. This is something we explicitly forbid as it runs against our definition of Agents which allows them only one thread-of-execution at a time.

Visibility of Changes Are the changes made (messages sent, environment modified) by an Agent which is updated during an Iteration-Order visible (during) *In-Iteration* or only *Post-Iteration* at the next Iteration-Order? More formally: do all the Agents $a_{n>i}$ which are updated after Agent a_i see the changes to the environment and messages sent to them by Agent a_i?

Repeatability Does the ABS has an external source of non-determinism which it cannot influence? If this is the case then we regard an update-strategy as *non-deterministic* and *deterministic* otherwise. It is important to distinguish between *external* and *internal* sources of non-determinism. The latter, coming from random-number generators, can be controlled using the same starting-seed thus leading to repeatability and deemed deterministic in this context. The former one are race-conditions due to concurrency, creating non-deterministic orderings of events which has the consequence that repeated runs may lead to different results with the same configuration thus rendering a ABS non-deterministic.

Having these properties identified we can now derive all meaningful and reasonable update-strategies which are possible in a general form in ABS. These update-strategies together with the properties will form the new terminology we propose to speak about update-strategies in ABS in general.

4 Update-Strategies

In this section we present the four general update-strategies which are possible in ABS. We give the list of all properties presented in the previous section, give a short description of the strategy and discuss their semantics and variations. We will discuss all details programming-language agnostic, give semantic meanings and interpretations of them and the implications selecting update-strategies for a model.

4.1 Sequential Strategy

Iteration-Order: Sequential
Global Synchronization: Yes
Thread of Execution: Shared
Message-Handling: Immediate or Queued
Visibility of Changes: In-Iteration
Repeatability: Deterministic

Description: This strategy has a globally synchronized 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.

Semantics: There is no source of randomness and non-determinism thus rendering this strategy to be completely deterministic in each step. Messages can be processed either immediately or queued depending on the semantics of the model. If the model requires to process the messages immediately the model must be free of potential recursions.

Variation: If the sequential iteration from Agent [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 still deterministic.
Using this strategy it is very easy to create the illusion of a local-time for each Agent by adding a random-offset to the global time for every Agent.
If one wants to have a very specific ordering, e.g. 'better performing' Agents first, then this can be easily implemented too by exposing some sorting-criterion and sorting the list of Agents after each Iteration.

4.2 Parallel Strategy

Iteration-Order: Parallel
Global Synchronization: Yes
Thread of Execution: Separate (or Shared)
Message-Handling: Queued
Visibility of Changes: Post-Iteration
Repeatability: Deterministic

Description: This strategy has a globally synchronized 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.

Semantics: 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.
It does not make a difference if the Agents are really computed in parallel or just sequentially, due to the isolation of actions, this has the same effect. Also it will make no difference if we iterate over the agents sequentially or randomly, the outcome *has to be* the same: the strategy 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. A workaround would be to implement the global environment as an Agent with which the non-environment Agents can communicate via messages thus we introduce an ordering but which is then sorted in a controlled order by an Agent, something which is not possible in the case of a passive, non-agent environment.

Variation: Using this strategy it is very easy to create the illusion of a local-time for each agent by adding a random-offset to the global time for every Agent.

4.3 Concurrent Strategy

Iteration-Order: Parallel
Global Synchronization: Yes
Thread of Execution: Separate
Message-Handling: Queued
Visibility of Changes: In-Iteration
Repeatability: Non-Deterministic

Description: This strategy has a globally synchronized 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 more general form of the Parallel Strategy: all Agents run at the same time but with actions becoming visible immediately.

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 when one ignores 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 has the very important influence that repeated runs with the same configuration of the Agents and the Model may lead to different results.

Variation: Using this strategy it is very easy to create the illusion of a local-time for each agent by adding a random-offset to the global time for every Agent.

4.4 Actor Strategy

Iteration-Order: Parallel
Global Synchronization: No
Thread of Execution: Separate
Message-Handling: Queued
Visibility of Changes: In-Iteration
Repeatability: Non-Deterministic

Description: This strategy has no globally synchronized 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.

Semantics: 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.
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 [10] for which I. Grief in [9] and W. Clinger in [4] developed semantics of different kinds. These works were very influential in the development of the concepts of Agents and and can be regarded as foundational basics for ABS.

Variation: It is important to understand that this strategy is the most general one as it allows to simulate all other strategies using synchronization.

5 Language Comparison [1]

In this section we give a brief overview of comparing the suitability of three fundamentally different languages to implement the update-strategies. We wanted to cover a wide range of different types of languages but didn't include 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. Also it was important to us that we tried to use the strenghts of each language as well as possible without abusing language constructs to recreate features it might seem to lack. An example would be to rebuild OO constructs in pure functional languages which would be a abuse of the language, something we explicitly avoided although it resulted in a few limitations as noted below.
For testing the suitability we selected a variety of simple models we implemented in each language with mostly all strategies. The selected models are *Heroes & Cowards*, *SIRS*, *Wildfire* and the *Prisoners Dilemma* mentioned in the introduction. We lack the space to explain all models but all are well known and can be easily found, looked up and understood on the Internet. They span different challenges to the ABS implementation: sending messages, accessing the environment, spawning new Agents, killing existing ones, discrete and continuous model. Although not implemented, we also claim that all the reference-models proposed in [12] and the StupidModel 1-16 by [15] can be faithfully capture using our new terminology.

5.1 Java

This language is included as the benchmark of object-oriented (OO) imperative languages as it is extremely popular in the ABS community and widely used in implementing their models and Environments. It comes with a comprehensive programming library, has nice object-oriented features and powerful synchronization primitives built in at language-level.

Ease of Use Being experienced Java-Programmers we found that implementing all the strategies was straight-forward and easy thanks to the languages features. Especially parallelism and concurrency is quite very easy due to elegant and powerful built-in synchronization primitives.

Benefits We experienced quite high-performance even for a large number of agents which we attributed to aliasing using references and side-effects. This prevents massive copying like Haskell but comes at the cost of explicit data-flow.

Deficits We couldn't identify something which absolutely didn't work. That's also why Java can be regarded as a very safe decision when looking for an appropriate language to use for implementing ABS.A downside is that one must take care when accessing memory in case of Parallel or Concurrent strategy. Due to the availability of aliasing and side-effects in the language and the type-system, it can't be guaranteed that access to memory happens only when its safe. Thus care must be taken when accessing references sent by messages to other Agents, accessing references to other Agents or the infrastructure of an Agent itself e.g. the message-box.
We found that implementing the Actor Strategy was not possible when using thousands of Agents because Java can't handle this number of threads. For implementing the Parallel and Concurrent ones we utilized the ExecutorService to submit as task for each Agent which runs the update and finishes then. The tasks are evenly distributed between the available threads using this service where the service is backed by the number of cores the CPU has. This approach does not work for the Actor Strategy because there an Agent runs constantly within its thread thus making it not possible to map to the concept of a task as this task would not terminate. The ExecutorService would then start n tasks (where n is the number of threads in the pool) and would not start new ones until those have finished, which will not occur until

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

the Agent would shut itself down. Also yielding or sleeping does not help either as not all threads are started but only n.

Natural Strategy We found that the Sequential Strategy with immediate message-handling is the most natural strategy to express in Java due to its heavy reliance on side-effects through references (aliases) and shared thread of execution. Also most of the models work this way and its thus a save decision to use Java.

5.2 Haskell

This language is included to put to test whether such a pure functional, declarative programming language is suitable for full-blown ABS. What distinguishes it is its complete lack of implicit side-effects, global data, mutable variables and objects. The central concept is the function into which all data has to be passed in and out explicitly through statically typed arguments and return values: data-flow is completely explicit.

Ease of Use Being beginners in Haskell we initially thought that it would be suitable at best for just implementing the Parallel Strategy due the inherent data-parallel nature of pure functional languages. After having implementing all strategies we had to admit that Haskell is very well suited to implement all of them faithfully. We think this stems from the facts that it has no implicit side-effects which reduces bugs considerably and results in very explicit data-flow.

Not having objects with data and methods which can call between each other meant, that we needed some different way of representing Agents. This was done using a struct-like type to carry data and a transformer function which would receive and process messages. This may seem to look like OO but it is not: Agents are not carried around but messages are sent to a receiver identified by an id.

Benefits We really enjoyed working in the extremely powerful static type-system. Although it seems to be restrictive in the beginning, when one gets used to it and knows how to use it for ones help, then it becomes rewarding. Our major point was to let the type-system prevent us from introducing side-effects. In Haskell this is only possible in code marked in its types as producing side-effects, so this was something we explicitly avoided and were able to do so throughout the whole implementation. This means a user of this approach can be guided by the types and is prevented from abusing them. Thus the lesson learned here is that *if one tries to abuse the types 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. Also we claim that when using Haskell one arrives at a much safer version in the case of Parallel or Concurrent Strategies than in Java.

Parallelism and Concurrency is a breeze in Haskell due to its complete lack of implicit side-effects. Adding hardware-parallel execution in the Parallel-Strategy required the adoption of only 5 lines of code and no change to the existing Agent-Code at all (e.g. no synchronization, as there are no implicit side-effects). For implementing the Concurrent Strategy we utilized the programming model of Software-Transactional-Memory (STM). The approach is that one optimistically runs Agents which introduce explicit side-effects in parallel where each Agent executes in a transaction and then to simply retry the transaction if another Agent has made concurrent side-effect modifications. This frees one from thinking in terms of synchronization and leaves the code of the Agent nearly the same as in the Sequential Strategy.
Spawning thousands of threads in the Actor Strategy is no problem in Haskell due to its lightweight handling of threads internal in the runt-time system, something which Java seems to be lacking. We have to note that each Agents needs to explicitly yield the execution to allow other Agent-threads to be sched-

uled, something when omitted will bring the system to a grind.

Deficits Performance is an issue. Our Haskell solution could run only about 2000 Agents in real-time with 25 updates per second as opposed to 50.000 in our Java solution, which is not very fast. It is important though to note, that being beginners in Haskell, we are largely unaware of the subtle performance-details of the language thus we expect to achieve a massive speed-up in the hands of an experienced programmer.

Another thing is that currently only homogeneous agents are possible and still much work needs do be done to capture large and complex models with heterogeneous agents. For this we need a more robust and comprehensive surrounding framework, which is already existent in the form of functional reactive programming (FRP). Our next paper is targeted on combining our Haskell solution with an FRP framework like Yampa (see further research).

Our solution so far is unable to implement the Sequential Strategy with immediate message-handling. This is where object-orientation really shines and pure functional programming seems to be lacking in convenience. A solution would need to drag the collection of all Agents around which would make state-handling and manipulation very cumbersome. In the end it would have meant to rebuild OO concepts in a pure functional language, something we didn't wanted to do. For now this is left as an open, unsolved issue and we hope that it could be solved in our approach with FRP (see future research).

Natural Strategy The most natural strategy is the Parallel-Strategy as it lends itself so well to the concepts of pure functional programming where things are evaluated virtually in parallel without side-effects on each other - something which resembles exactly the semantics of the Parallel Strategy. We argue that the Concurrent Strategy is also very natural formulated in Haskell due to the availability of STM, something only possible in a language without implicit side-effects as otherwise retries of transactions would not be possible.

5.3 Scala with Actors

This multi-paradigm functional language is included to test the usefulness of the Actor Strategy for implementing ABS. The language comes with an Actor-library inspired by [2] and resembles the approach of Erlang which allows a very natural implementation of the strategy.

Ease of Use We were completely new to Scala with Actors although we have some experience using Erlang. We found that the language has some very nice mixed-paradigm features which allow to program in a very flexible way without inducing too much restrictions on one.

Benefits Implementing Agent-behaviour is extremely convenient, especially for simple state-chart Agents. The Actor-language has a built-in feature which allows to change the behaviour of an Agent on message reception where the Agent then simply switches to a different message-handler, allowing elegant implementation of state-charts.

Performance is very high. We could run simulations in real-time with about 200.000 Agents concurrently, something the run-time system easily manages. Also it is very important to note that one can use the framework Akka to build real distributed systems using Scala with Actors so there are potentially no limits on the size and complexity of the models and number of agents one wants to run with it.

Deficits Care must be taken not to send references and mutable data, which is still possible in this mixed-paradigm language.

Natural Strategy The most natural strategy would be of course the Actor Strategy and we only used this Strategy in this language to implement our Models. Note that the Actor Strategy is the most general one and would allow to capture all the other strategies using the appropriate synchronization mechanisms.

6 Results

In this section we will revisit the Prisoners Dilemma model of [14] and the Heroes & Cowards model of [20] and present results simulating both with our four update-strategies.

6.1 Prisoners Dilemma

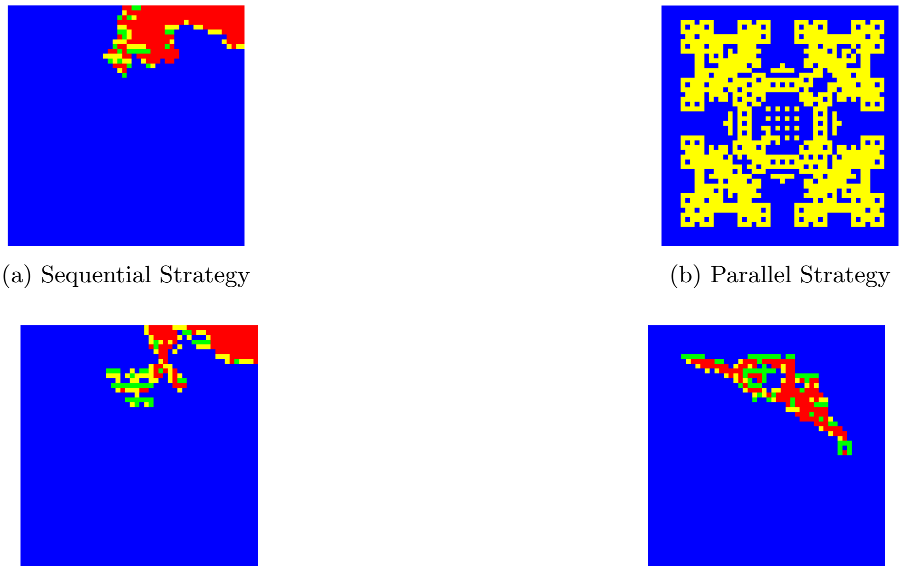
It is immediate clear that, when looking at figure 2 the update-strategy which reflects the semantics of the model is the Parallel Strategy as all others clearly fail to reproduce the pattern as predicted by the model. Thus we can say only the Parallel Strategy is suitable to simulate this model and that only this Strategy is the correct one.

The reason why the others fail to reproduce the pattern is due to the non-parallel and unsynchronized way that information spreads through the grid: in the Sequential Strategy the Agents further ahead in the queue play the game earlier and influence the neighbourhood thus Agents in the neighbourhood which play the game later find an already changed environment/messages and act thus based upon these informations. This is not the case in the Parallel version: all Agents play the game on the frozen state of the previous step and the outcome of each Agents game will only be visible in the next step. In the Concurrent and Actor Strategy the Agents run in parallel but changes are visible immediately and concurrently, thus leading to the same non-structural patterns as in the Sequential Strategy.

Note that the Concurrent and Actor Strategy produce different results on every run due to the inherent non-deterministic event-ordering introduce by concurrency. Also note that it is not possible to calculate 45 steps for the Actor Strategy as it lacks the Global Synchronization property. To arrive at a relative comparative result we just waited until the first Agent arrives at a local time of 45 and then rendered the result.

6.2 Heroes & Cowards

One starts with a crowd of Agents where each Agent is positioned randomly in a continuous 2D-space. Each of the Agents then selects randomly one friend and one enemy (except itself in both cases) and decides with a given probability whether the Agent acts in the role of a "Hero" or a "Coward" - friend, enemy and role don't change after the initial set-up. Now the simulation can start: in each step the Agent will move a given distance towards a given point. If the Agent is in the role of a "Hero" this point will be the half-way distance between the Agents friend and enemy - the Agent tries to protect the friend from the enemy. If the Agent is acting like a "Coward" it will try to hide behind the friend also the half-way distance between the Agents friend and enemy, just in the opposite direction. The world this model is situated in is restricted by borders in the form of a rectangle: the agents cannot move out of it and will be clipped against the border if the calculation would end them up outside. Note that this simulation is determined by the random starting positions, random friend & enemy selection, random role selection and number of agents. Note also that during the simulation-stepping no randomness is mentioned in the model and given the initial random set-up, the simulation-model is completely deterministic.



(a) Sequential Strategy

(b) Parallel Strategy

(c) Concurrent Strategy

(d) Actor Strategy

Figure 2: Haskell implementation of Prisoner-Dilemma game as in [11] on a 50x50 grid after 45 steps.

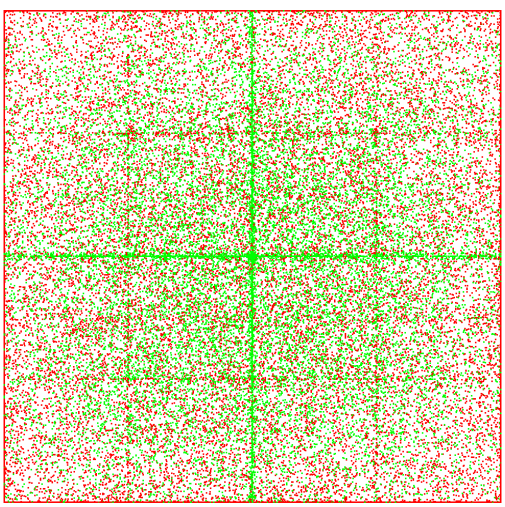


Figure 3: Emergent cross-pattern forming in all four update-strategies using 100.000 Agents with 25% Heroes. Picture taken from running the Concurrent Strategy for 500 steps in our Java implementation.

Although the individual Agent-positions of runs with the same configuration differ between update-strategies we experienced the forming of the emergent cross-pattern as seen in figure 3 in all four update-strategies. Thus we can conclude that the Heroes & Cowards model seems to be more robust to the selection of its update-strategy and that its emergent property - the formation of the cross - is stable under differing update-strategies. One would not see a difference between the different strategies thus only one picture was included. Note that to test the Actor Strategy with a this high number of Agents we used our implementation in Scala with Actors as Java is not able to have this high number of threads and our Haskell implementation suffers from performance issues, thus resorting to Scala with Actors. The results were nearly the same there, showing the big green emergent cross-pattern in the center but lacking the smaller red crosses in each section, something we attribute to the local-time of each Agent and the relativity of observing the simulation.

7 Conclusion

In this paper we presented the four general update-strategies for an ABS and discussed their implications. Again we cannot stress enough that selecting the correct update-strategy is of most importance and must match the semantics of the model one wants to implement.

We also argued that the ABS community needs a unified terminology of speaking about update-strategies otherwise confusions arise and reproducibility suffers. We proposed such a unified terminology on the basis of the general update-strategies and hope it will get adopted.

To put our theoretical considerations to a practical test we implemented them in three very different kind of languages to see how each of them performed in comparison with each other in implementing the update-strategies. To summarize, we can say that Java is the gold-standard due to convenient synchronization primitives built in the language. Haskell really surprised us as it allowed us to faithfully implement all strategies equally well, something we didn't anticipate in the beginning of our research. We hope that our work convinces researches and developers in the field of ABS to give Haskell a try and dig deeper into it, as we feel it will be highly rewarding.

If one can live with the non-determinism of Scala with Actors it is probably the most interesting and elegant solution to implement ABS. We attribute this to the closeness of Actors to the concept of Agents, the powerful concurrency abstraction and language-level support.

8 Further Research

8.1 Stability of Emergent Properties

It would be interesting to look deeper into the questions why some models require a given update-strategy and why some emergent properties are invariant under different update-strategies. We leave this open for further research as it would be clearly out of scope of this paper to develop such a theoretical model of emergent behaviour under different update-strategies.

8.2 Functional Reactive Programming

[13] discuss using functional programming for discrete event simulation (DES) and mention the paradigm of Functional Reactive Programming (FRP) to be very suitable to DES. We were aware of the existence of this paradigm and have experimented with it using the library Yampa, but decided to leave that topic to a side and really keep our implementation clear and very basic. The next step would be to fusion ABS, which can be understood as a variant of DES, with Yampa thus leveraging both approaches from which we hope to gain the ability to develop much more complex models with heterogeneous agents.

8.3 Functional Model-Specification Language

After showing that Haskell is a very attractive alternative to existing OO approaches in implementing ABS we are interested whether the declarative power of the pure functional language can be utilized to write specifications for simple ABS models. Such a language would be equals or very close to the program-code thus eliminate the gap between specification and programming.

8.4 Actor Model in ABS

Although we showed that the Act-Strategy implemented in Scala with Actors can implement very different kind of Models we barely scratched the surface. There already exists research using the Actor Model for ABS in the context of Erlang [19], [7], [8], [16] but we find that the Actor Model should get more attention in ABS. We think that this research-field is nowhere near exhaustion and we hope that more research is going into this topic as we assume that the Actor-Model has a bright future ahead due to the ever increasing availability of massively parallel computing machinery.

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] De Jong, T. Suitability of Haskell for Multi-Agent Systems. Tech. rep., University of Twente, 2014.

[7] Di Stefano, A., and Santoro, C. Using the Erlang Language for Multi-Agent Systems Implementation. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology* (Washington, DC, USA, 2005), IAT '05, IEEE Computer Society, pp. 679–685.

[8] Di Stefano, A., and Santoro, C. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. Tech. rep., 2007.

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

[10] 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.

[11] 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.

[12] Isaac, A. G. The ABM template models: A reformulation with reference implementations.

[13] Jankovic, P., and Such, O. Functional Programming and Discrete Simulation. Tech. rep., 2007.

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

[15] Railsback, S., Lytinen, S., and Grimm, V. StupidModel and Extensions: A Template and Teaching Tool for Agent-based Modeling Platforms, Dec. 2005.

[16] Sher, G. I. *Agent-Based Modeling Using Erlang Eliminating The Conceptual Gap Between The Programming Language & ABM.* 2013.

[17] Sorokin, D. Aivika 3: Creating a Simulation Library based on Functional Programming, 2015.

[18] Sulzmann, M., and Lam, E. Specifying and Controlling Agents in Haskell. Tech. rep., 2007.

[19] Varela, C., Abalde, C., Castro, L., and Gulías, J. On Modelling Agent Systems with Erlang. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2004), ERLANG '04, ACM, pp. 65–70.

[20] Wilensky, U., and Rand, W. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo.* MIT Press, 2015.

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

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