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

Jonathan Thaler

February 17, 2017

Abstract

When developing a model for an Agent-Based Simulation (ABS) it is very important to select the update-strategy which reflects the semantics of the model because simulation results can vary vastly across different update-strategies. This awareness, we claim, is still lacking in the field of ABS. In this paper we derive general properties of ABS and use them to classify all update-strategies possible in ABS. This will allow implementers and researchers in this field to use a general terminology, removing ambiguities when discussing ABS and their models. We prove that our classification works by providing case studies in implementing two different kind of games in three very different programming languages Java, Haskell and Scala and compare their suitability to implement the strategies. The major finding is that depending on the kind of game one can expect very different results when selecting different update-strategies.

Keywords

Agent-Based Simulation, Parallelism, Concurrency, Haskell, Actors, Prisoners Dilemma, Heroes and Cowards

1 Introduction

In this paper we are looking at two different kind of games to show the differences update-strategies can make in ABS and supporting our main message that *when developing a model for an ABS it is of most importance to select the right update-strategy which reflects and supports the corresponding semantics of the model*. As we will show due to conflicting ideas about update-strategies this awareness is yet still under-represented in the field of ABS and is lacking a systematic treatment. As a remedy we undertake such a systematic treatment in proposing a new terminology by identifying properties of ABS and deriving all possible update-strategies. The by-product is a framework to talk in a unified form about this very important matter, so to enable researchers and implementers to talk about it in a common way, enabling better discussions, same understanding and better reproduceability and continuity in research. The two games we use are discrete and continuous games where in the former one the agents act synchronized at discrete time-steps whereas in the later one they act continuously in continuous time. We show that in the case of simulating the discrete game the update-strategies have a huge impact on the final result whereas our continuous game seems to be stable under different update-strategies. Because the selection of an update-strategy has profound implications for the implementation of an ABS we investigate the three different programming-paradigms of *object-orientation* (OO), *pure functional* and *multi-paradigm* in the form of the programming languages Java, Haskell and Scala in their suitability of implementing each update-strategy. As it turns out the paradigms can't capture all the update-strategies equally well thus one should be careful when selecting the implementation language for the ABS, reflecting its suitability for implementing the selected updates-strategy. The contribution of this paper is:

- Present general properties of ABS.
- Derive update-strategies from these properties.
- Establish a general terminology of talking about these update-strategies.
- Compare the three programming languages Java, Haskell and Scala in regard of their suitability to implement each of these strategies.

2 Background

In this section we present the background of our research. We define our term and understanding of *agent* and ABS and how we understand and use it in this paper, then we will give a description of the two kind of games which were the motivators for our research and case-studies and finally we will present related work.

2.1 Agent-Based Simulation

We understand ABS as 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. So 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 an environment, able to spawn new agents and interacting with other agents in a network of neighbours by exchange of messages [20]. It is important to note that we focus our understanding of ABS on a very specific kind of agents where the focus is on communicating entities with individual, localized behaviour from out of which the global behaviour of the system emerges. We informally assume the following about our agents:

- They are uniquely addressable entities with 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 an environment they are situated in.

An implementation of an ABS must 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 stimulus, 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 stimulus. This allows the agent then to perceive time and become pro-active depending on time. So 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 where there are two ways of feeding time-flow into an agent. Either it has its own thread-of-execution or the system creates the illusions of its 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 similarity 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.

2.2 A discrete game: Prisoners Dilemma

As an example of a discrete game we use the *Prisoners Dilemma* as presented in [14]. In the prisoners dilemma two players play a game where in each step a player can either choose to cooperate or defect receiving a pay-off. There are four possible pay-offs: if both players cooperate both receive R; if one player defects and the other cooperates the defector receives T and the cooperator S; if both defect both receive P where $T > R > P > S$. In the version of [14] NxN agents are arranged on a 2D-grid where every agent has 8 neighbours. Agents don't have a memory of the past and have one of two roles: either cooperator or defector. In every step an agent plays the game with all its neighbours, including itself (something which was not explicitly mentioned in the paper but if omitted, will not lead to their results) and sums up the pay-off. After the pay-off sum is calculated the agent changes its role to the role of the agent with the highest pay-off within its neighbourhood (including itself). The authors attribute the following pay-offs: S=P=0, R=1, T>b, where b>1. They showed that when having a grid of only cooperators with a single defector at the center, the simulation will form beautiful structural patterns as shown in figure 1.
In [10] the authors showed that the results of simulating the *Prisoners Dilemma* as above depends on a very specific strategy of iterating the simulation and show that the beautiful patterns seen in figure 1 will not form when selecting a different update-strategy. They introduced the terms of synchronous and asynchronous updates and define synchronous to be as

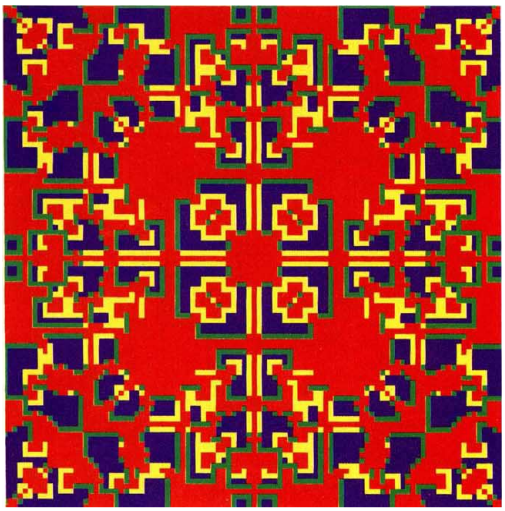


Figure 1: Patterns formed by playing the *Prisoners Dilemma* game on a 99x99 grid with $1.8 < b < 2$ after 217 steps. Blue are cooperators, red are defectors, yellow are cooperators which were defectors in the previous step, green are defectors which were cooperators in the previous step. Picture taken from [14].

agents being updated in unison and asynchronous where one agent is updated and the others are held constant. Only the synchronous updates are able to reproduce the results. Although the authors differentiated between the two strategies, their description still lacks precision and detail, something we will provide in this paper. Although they didn't publish their work in the field of ABS, it has general implications for ABS as well which can be generalized in the main message of our paper as emphasised in the introduction. We will show that there are more than two update-strategies and will give results of simulating this discrete game using all update-strategies.

2.3 A continuous game: Heroes & Cowards

As an example for a continuous game we use the *Heroes & Cowards* game introduced by [19]. In this game one starts with a crowd of agents where each agent is positioned randomly in a continuous 2D-space which is bounded by borders on all sides. Each of the agents then selects randomly one friend and one enemy (except itself) 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. In each step the agent will move a small distance towards a target point. If the agent is in the role of a hero this target 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. Note that this simulation is determined by the random starting positions, random friend and enemy selection, random role selection and number of agents. Note also that during the simulation-stepping no randomness is incurred and given the initial random set-up, the simulation-model is completely deterministic. As will be shown later the results of simulating this model are invariant under different update-strategies.

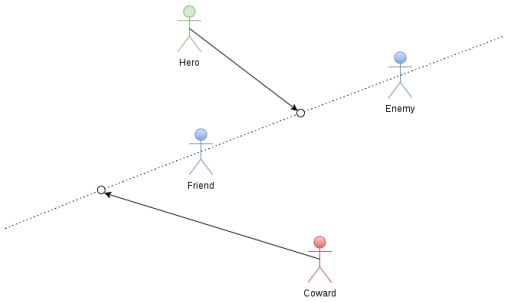


Figure 2: A conceptual diagram of the *Heroes & Cowards* game. Hero (green) and coward (red) have the same agents as friend and enemy but act different: the hero tries to move in between the friend and enemy whereas the coward tries to hide behind its friend.

2.4 Related Research

[13] 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 haven't mapped our ideas to GPUs we explicitly include an approach for data-parallelism which, we hypothesize, can be utilized to roughly map their approach onto our terminology.

[2] sketch a minimal ABS 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 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, our focus is very different from theirs and approaches ABS in a more general and comprehensive way.

[4] 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 asyn-

chronous 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.

[21] presents 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.

[12] 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 amount of research on using Haskell in the field of ABS has been moderate so far. Though there exist a few papers which look into Haskell and ABS [5], [17], [12] they focus primarily on how to specify agents. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [16]. It also comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. This papers is investigating Haskell in a different way by looking into its suitability in implementing update-strategies in ABS, something not looked at in the ABS community which presents an original novelty as well.

There already exists research using the Actor Model [1] for ABS in the context of Erlang [18], [6], [7], [15] but we feel that they barely scratched the surface. We want to renew the interest in this direction of research by incorporating Scala with using the Actor-library in our research because we will show that one update-strategy maps directly to the Actor Model.

3 A new terminology

When looking at related work, 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. 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. Having these properties identified we then 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. For each strategy we give the list of all properties, 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. A summary of all update-strategies and their properties are given in table 1.

3.1 ABS Properties

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 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, rendering a ABS non-deterministic.

3.2 ABS Update-Strategies

3.2.1 Sequential Strategy

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.

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

Semantics: There is no source of randomness and non-determinism, 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 implying it is 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 collection of agents after each Iteration.

3.2.2 Parallel Strategy

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.

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

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, violating the *happens at the same time* idea. To solve this, we require different semantics for accessing the environment in this strategy. We introduce a *global* environment which is made up of the set of *local* environments. Each local environment is owned by an agent so 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*. 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 introducing an ordering but which is then sorted in a controlled way 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.

3.2.3 Concurrent Strategy

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. So this strategy can be understood as a more general form of the *parallel strategy*: all agents run at the same time but acting concurrently.

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

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. 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 arriving at non-deterministic solutions. This has the very important consequence 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.

3.2.4 Actor Strategy

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.

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

Semantics: It is of most importance to note that information and also time in this strategy is always local to an agent as each agent progresses in its own speed through the simulation. 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. Problems can occur though if the observer-agent can't process the update-messages fast enough, resulting in a congestion of its message-queue. The concept of Actors was proposed by C. Hewitt in 1973 in his work [9] for which I. Grief in [8] and W. Clinger in [3] 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: 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.

4 Programming paradigms and ABS

In this section we give a brief overview of comparing the suitability of three fundamentally different languages to implement the different update-strategies. We wanted to cover a wide range of different types of languages and putting emphasis on each languages strengths 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. We implemented both the *Prisoners Dilemma* game on a 2D grid and the *Heroes & Cowards* game in all three languages with all four update-strategies. [1]

4.1 OO: 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 frameworks. It comes with a comprehensive programming library and powerful synchronization primitives built in at language-level.

Ease of Use We found that implementing all the strategies was straight-forward and easy thanks to the languages features. Especially parallelism and concurrency is convenient to implement due to elegant and powerful built-in synchronization primitives.

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

Table 1: Update-Strategies in ABS

	Sequential	**Parallel**	**Concurrent**	**Actor**
Iteration-Order	Sequential	Parallel	Parallel	Parallel
Global-Sync	Yes	Yes	Yes	No
Thread	Shared	Separate	Separate	Separate
Messaging	Immediate	Queued	Queued	Queued
Visibility	In	Post	In	In
Repeatability	Yes	Yes	No	No

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

Deficits 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 it can't be guaranteed by Java's type-system that access to memory happens only when its safe. So 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 a 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 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 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 making Java a save choice for implementing ABS.

4.2 Pure functional: Haskell

This language is included to put to test whether 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. For a nice overview on the features and strengths of pure functional programming see the classical paper [11].

Ease of Use We initially thought that it would be suitable best for implementing the *parallel strategy* only 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 fact that it has no implicit side-effects which reduces bugs considerably and results in completely 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 Haskell has a very powerful static type-system which seems to be restrictive in the beginning but when one gets used to it and knows how to use it for ones support, 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. In essence, the lesson learned here is *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 extremely convenient to implement 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 scheduled, 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 so 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 OO 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 with slightly more effort, 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.

4.3 Multi-paradigm: Scala

TODO: implement other strategies TODO: much shorter than the other two, because not implemented all and not so much to say
This multi-paradigm functional language which sits in-between Java and Haskell and is included to test the usefulness of the *actor strategy* for implementing ABS. The language comes with an Actor-library inspired by [1] 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 which was of great use. We found that the language has some very powerful 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-driven 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-dependent behaviour.
Performance is very high. We could run simulations in real-time with about 200.000 agents concurrently, thanks to the transparent handling in the run-time system. 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 to 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.

5 Case-Studies

In this section we present two case-studies in simulating the *Prisoners Dilemma* and *Heroes & Cowards* games for discussing the effect of using different update-strategies. As already emphasised both are of different nature. The first one is a discrete game, played at discrete time-steps. The second one is a continuous game where each agent is continuously playing. This has profound implications on the simulation results as will is shown below. The results show that depending on the type of model the results can be very different when using different update-strategies as happening in the case of *Prisoners Dilemma*. Results of other models seem to be stable under varying update-strategies as is the case with *Heroes & Cowards*.

5.1 Prisoners Dilemma

The results as seen in the left column of figure 3 were created using the Haskell implementation using the same configuration as reported in the original paper: a 99x99 grid with all cooperators except one defector at the center, running for 217 steps. When comparing the pictures with the one from the original paper seen in figure 1 the only update-strategy which is able to reproduce the matching result is the *parallel strategy* - all the other clearly fail to reproduce the pattern. From this we can deduce that only the *parallel strategy* is suitable to simulate this model because only that strategy is the one which renders the results of the original paper, meaning it is the 'correct' strategy for this model. The reason why the other strategies fail to reproduce the pattern is due to the non-parallel and unsynchronized way that information spreads through the grid. In the *sequential*

Table 2: Language Comparisons

	Java	Haskell	Scala
Sequential	++	–	? (+)
Parallel	+	++	? (+)
Concurrent	+	+	? (+)
Actor	–	+	++

strategy the agents further ahead in the queue play the game earlier and influence the neighbourhood so agents which play the game later find already messages from earlier agents in their queue thus acting differently based upon these informations. In the *concurrent* and *actor strategy* the agents run in parallel but changes are visible immediately and concurrently, leading to the same non-structural patterns as in the *sequential* one. This is not the case in the *parallel strategy* where, in every step, all agents play the game at the same time based on the frozen state of the previous step, leading to a synchronized update as required by the model. Note that the *concurrent* and *actor strategy* produce different results on every run due to the inherent non-deterministic event-ordering introduce by concurrency. As there are no global time-steps in the *actor strategy*, to calculate 45 steps we just waited until the first agent arrives at a local time of 45 and then rendered the result.

5.2 Heroes & Cowards

The results as seen in the right column of figure 3 were created using the Java and Scala implementations with 100.000 agents where 25% of them were heroes. Although the individual agent-positions of runs with the same configuration differ between update-strategies the cross-patterns are forming in all four update-strategies. For the patterns to emerge it is important to have significant more cowards than heroes and to have more than 75.000 agents - we went for 100.000 because then the patterns are really prominent. We can conclude that the *Heroes & Cowards* model seems to be robust to the selection of its update-strategy and that its emergent property - the formation of the cross - is stable under differing strategies. To test the *actor strategy* with this high number of agents we used our implementation in Scala with the Actor-library as Java is not able to have this high number of threads and our Haskell implementation suffers from performance issues.

6 Conclusion and future research

In this paper we presented general properties of ABS, derived four general update-strategies and discussed their implications. Again we cannot stress enough that selecting the right update-strategy is of most importance and must match the semantics of the model one wants to simulate. 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. We put our theoretical considerations to a practical test by implementing case-studies using 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. We explicitly avoided using the functional reactive programming (FRP) paradigm to keep our solution simple but could only build simple models with homogeneous agents. The next step would be

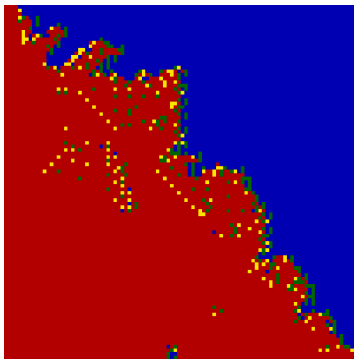
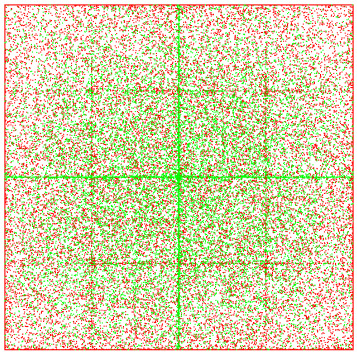
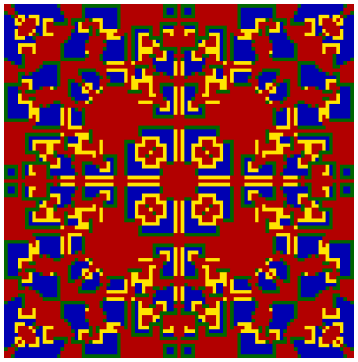
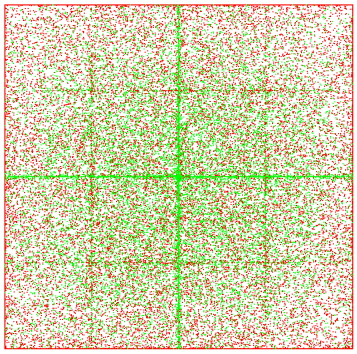
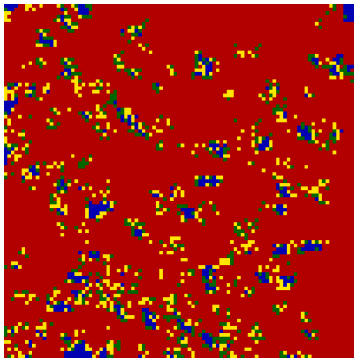
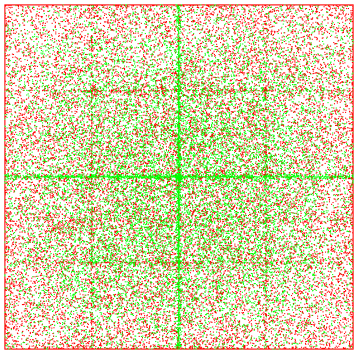
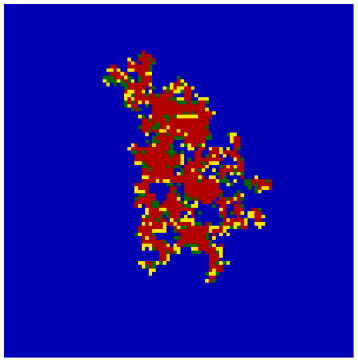
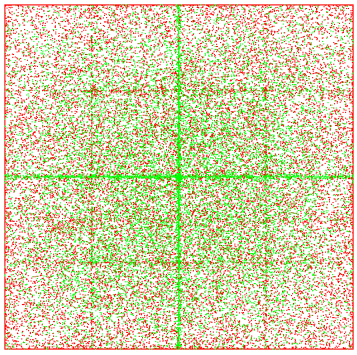
	Prisoners Dilemma	Heroes & Cowards
sequential strategy	 (a)	 (b)
parallel strategy	 (c)	 (d)
concurrent strategy	 (e)	 (f)
actor strategy	 (g)	 (h)

Table 3: Effect on results simulating the Prisoners Dilemma and Heroes & Cowards with all four update-strategies.

to fusion ABS with FRP using the library Yampa for leveraging both approaches from which we hope to gain the ability to develop much more complex models with heterogeneous agents. If one can live with the non-determinism of Scala with the Actors-library 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. We barely scratched the surface on this topic 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. We showed that the the *Prisoners Dilemma* game on a 2D-grid can only be simulated correctly when using the *parallel strategy* and that the other strategies lead to a break-down of the emergent pattern reported in the original paper. On the other hand using the *Heroes & Cowards* game we showed that there exist models whose emergent patterns exhibit a stability under varying update-strategies. Intuitively we can say that this is due to the nature of the model specification which does not require specific orderings of actions but it would be interesting to put such intuitions on firm theoretical grounds.

References

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

[2] 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.

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

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] Hughes, J. Why Functional Programming Matters. *Comput. J. 32*, 2 (Apr. 1989), 98–107.

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

[13] Lysenko, M., D'souza, R., and Rahmani, K. *A Framework for Megascale Agent Based Model Simulations on the GPU.* 2008.

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

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

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

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

[18] 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.

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

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

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