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

Jonathan Thaler

January 26, 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 the 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.

1 Introduction

In the paper of [8] the authors showed that the results of the simulation of the classic prisoners-dilemma on a 2D-grid reported in [10] depends on a a very specific strategy of iterating this simulation and show that the beautiful patterns as reported by [10] will not form when selecting a different iteration-strategy. Although the authors differentiated between two strategies, their description still lacks precision and generality which we will try to repair in this paper. Although they too discussed philosophical aspects of choosing one strategy over the other, they lacked to generalize their observation. We will do so in the central message of our paper by stressing that when doing Agent-Based Simulation & Modelling (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 all the general properties and deriving update-strategies which are possible in an ABS.
- Developing a new, general terminology of talking about them.
- Giving the semantic interpretation and meaning of each of them.
- Comparing the 3 programming languages Java, Haskell and Scala with Actors in regard of their suitability to implement each of these strategies.

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

2 Related Research

Already noted in the introduction, [8] 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 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 into our terminology.

[3] sketch a minimal agent-framework in Haskell which is very similar in the basic structure of ours. This proofs that this approach, very well developed in ABS, seems to be a very natural one also to apply to Haskell. Their focus is more on economic simulations and instead of iterating a simulation with a global time, their focus is on how to synchronize agents which have internal, local transition times. Although their work uses Haskell as well, this does not diminish the novelty of our approach using Haskell because our focus is a very different from them and approaches ABS in a more general and comprehensive sense.

[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 in same time intervals.

[12] 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-model. 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 vs. synchronous updates but the very semantic and technical details are unclear and not described very precisely. In the section about a new terminology we will address this issue and will put forward a proposal of how to fit these differences into our update-strategies and speak in a consistent way about them. there is no consistent use and understanding of it in the literature of ABS. Also it is imprecise and lacking important details which are of importance for the semantics of a Model. Thus we argue that there is still a lack of awareness about the influence of results due to the lack of important properties.

3 Background

ABM/S is a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is of knowledge. 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 unique pro-active unit, able to spawn new Agents, interacting with other Agents in a network of neighbours by exchange of messages which are situated in a generic environment [11]. 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 three fundamental problems:

1. Source of pro-activity
   How can an Agent initiate actions without the external stimuli of messages?

2. Message-Visibility
   When is a message m, visible to Agent B, processed by it?

3. Semantics of Message-Delivery
   When is a message m, sent by Agent A to Agent B, visible to B?

TODO: add "own thread of execution" or "shared thread of execution". still this does not allow us to generalize the notion of global time because in PAR and CON we have own threads of execution but global time. this brings us to: TODO: synchronized vs asynchronous iteration.
In computer systems, pro-activity, the ability to initiate actions on its own without external stimuli, is only possible when there is some internal stimuli, most naturally represented by some generic notion of monotonic increasing time-flow. This can either be some physical real-time system-clock which counts the milliseconds since 1970 (thus binding the time-flow of the system to the one of the 'real time') or a virtual simulation-clock which is just a monotonic increasing natural number. As we are in a discrete computer-system, this time-flow must be discretized as well in discrete steps and each step must be made available to the Agent, acting as the internal stimuli. This allows the Agent then to perceive time and become pro-active depending on time (NOTE: we could argue that this is not really pro-activity because it depends always on time, but there is really no other way of doing this in our current implementation of computer-systems.). Independent of the representation of the time-flow we have the two fundamental choices whether the time-flow is local to the Agent or whether it is a system-global time-flow.

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

Basically we can say that we want to process a message as soon as it is visible to us but this is not how real computer-systems can work. In a real system each Agent would have a message-box into which the messages are posted so the Agent can then check its mail-box for new messages. The question is then when the Agent is going to poll for new messages? Clearly what we need is a recurring, regular trigger which allows the Agent to poll the mail-box and process all queued messages. We argue that the most natural approach is to bind this trigger to the time-flow step which provides pro-activity. Note that all explicit communication between Agents (there could be implicit communication through the environment) is message-based where messages are only processed at specified times in execution which is the time-step of an Agent. Thus a message sent to another Agent is not executed by the Agent at the sending time but queued and executed at the next internal stimulus: the next time-step. This prevents recursive calls due to an agent could never know who initiated the message-chain, something an agent does not want to care for. TODO: maybe this is the difference between sync and async meant by some: sync is if messages are executed immediately, async is if message-execution is postponed

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

4 A new terminology

In this section we will present a new terminology to speak about update-strategies in ABS by presenting

the properties characterizing the dimensions and details of update-strategies and then derive all possible forms of update-strategies in a general form. We will discuss all details programming-language agnostic, give semantic meanings and interpreations of them and the implications selecting update-strategies for a model.
To develop a new terminology, we propose to abandon the notion of synchronous and asynchronous updates and put forward the following properties for characterizing the dimensions of update-strategies:

Iteration-Order Is the collection of Agents updated *sequential* with one Agent updated after the other or are all agents updated in *parallel*?

Iteration-Synchronization Is a full Iteration over the collection of Agents happening at *synchronous* points in time or does it happen *asynchronous*?

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.

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. TODO: elaborate on this: when doing parallel processing we really must enforce this constraint otherwise Agents could act concurrently, something which we want explicitly forbid

Visibility of Changes Are the changes made (messages sent, environment modified) by an Agent which is updated during an Iteration visible (during) *In-Iteration* or only *Post-Iteration* at the next Iteration? 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 May repeated runs lead to different results with the same configuration? If this is the case then the update-strategy is regarded as *non-deterministic* and *deterministic* otherwise.

4.1 Update-Strategies

4.1.1 Sequential Strategy

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

Description: This strategy has a global time-flow and in each time-step iterates through all the agents and updates one Agent after another. Messages sent and changes to the environment made by Agents are visible immediately.
Semantics: More formally, we assume that, given the updates are done in order of the index $i = [1..n]$, then Agents $a_{n>i}$ see the changes to environment and messages sent to them by Agent a_i. Note that there is no source of randomness and non-determinism thus rendering this strategy to be completely deterministic in each step.
Extensions: If the sequential iteration from 1..n imposes an advantage over the Agents further ahead or behind in the queue (e.g. if it is of benefit when making choices earlier than others in auctions or later when more information is available) then one could use random-walk iteration where in each time-step the agents are shuffled before iterated. Note that although this would introduce randomness in the model the source is a random-number generator thus reproduce-able.
TODO: what if we want to introduce a specific ordering? this can be easily implemented with comparators e.g. in java. but how is this in haskell?

4.1.2 Parallel Strategy

Iteration-Order: Parallel
Iteration-Synchronization: Synchronous
Thread of Execution: Separate
Message-Handling: Queued
Visibility of Changes: Post-Iteration
Repeatability: Deterministic

Description: This strategy has a global time-flow and in each time-step iterates through all the agents and updates all Agents in parallel. Messages sent and changes to the environment made by Agents are visible in the next global step. We can think about this strategy that all Agents make their moves at the same time.

Environment: If one wants to change the environment in a way that it would be visible to other Agents this is regarded as a systematic error in this strategy. First it is not logical because all actions are meant to happen at the same time and also it would implicitly induce an ordering thus violating the *happens at the same time* idea. Thus we require different semantics for accessing the environment in this strategy. We introduce thus a *global* environment which is made up of the set of *local* environments. Each local environment is owned by an Agent thus there are as many local environments as there are Agents. The semantics are then as follows: in each step all Agents can *read* the global environment and *read/write* their local environment. The changes to a local environment are only visible *after* the local step and can be fed back into the global environment after the parallel processing of the Agents.

Semantics: It does not make a difference if the Agents are really computed in parallel or just sequentially, due to the semantics of changes, this has the same effect. In this case it will make no difference how we iterate over the agents (sequentially, randomly), the outcome *has to be* the same - it is event-ordering invariant as all events/updates happen *virtually* at the *same time*. Thus if one needs to have the semantics of writes on the whole (global) environment in ones model, then this strategy is not the right one and one should resort to one of the other strategies.

4.1.3 Concurrent Strategy

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

Description: This strategy has a global time-flow and in each time-step iterates through all the agents and updates all Agents in parallel but all messages sent and changes to the environment are immediately visible. Thus this strategy can be understood as a mix of Seq and Par: all Agents run at the same time with actions becoming immediately visible.

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

4.1.4 Actor Strategy

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

Description: This strategy has no global time-flow but all the Agents run concurrently in parallel, with their own local time-flow. The messages and changes to the environment are visible as soon as the data arrive at the local Agents - this can be immediately when running locally on a multi-processor or with a significant delay when running in a cluster over a network. Obviously this is also a non-deterministic strategy and repeated runs with the same Agent and Model-configuration may (and will) lead to different results.

Locality of Information: It is of most importance to note that information and thus also time in this strategy is always local to an Agent as each Agent progresses in its own speed through the simulation. Thus in this case one needs to explicitly *observe* an Agent when one wants to e.g. visualize it. This observation is then only valid for this current point in time, local to the observer but not to the Agent itself, which may have changed immediately after the observation. This implies that we need to sample our Agents with observations when wanting to visualize them, which would inherently lead to well known sampling issues. A solution would be to invert the problem and create an Observer-Agent which is known to all Agents where each Agent sends a *'I have changed'* message with the necessary information to the observer if it has changed its internal state. This also does not guarantee that the observations will really reflect the actual state the Agent is in but is a remedy against the notorious sampling.

Semantics: This is the most general one of all the strategies as it can emulate all the others by introducing the necessary synchronization mechanisms and Agents. Also this concept was proposed by C. Hewitt in 1973 in his work [7] where upon I. Grief in [6] and W. Clinger in [4] developed semantics of different kinds. These works were very influential in the development of the Agent-Term and concept and can be regarded as foundational basics for ABM/S.

4.2 Mapping existing terminology

TODO: should be check for reference-models and look if all can be implemented: [9] and StupidModel 1-16

Although all definitions mentioned in the related research section have different semantics and implications, our framework faithfully captures all of them:

TODO: rework: don't cite the works but recap what they mean with sync and async

When following [12] we use *Seq*, *Par* or *Con* and the synchronous updates will happen in the pro-active internal stimulus and the asynchronous by reacting to and sending messages.

when following [5] we use *Seq*, *Par* or *Con* and the synchronous updates happen by providing the same global time-delta to all agents, which means all agents have global time and advance at the same speed. The asynchronous version would be to provide each agent with a different, random time-delta, rendering the time of the agent local instead of globally synchronized. When taking this into account we can argue that *Seq*, *Par* and *Con* are, as described above, synchronous update-strategies and only *Act* is truly asynchronous due to the locality of time and information. We can though make *Seq*, *Par* and *Con* asynchronous too if we iterate globally as specified but instead of feeding each Agent the global time which advanced by some constant delta since the previous step, we feed each Agent a local time by introducing random time-deltas drawn individually in a given range for each Agent. This would resemble the locality of the Act strategy without introducing the non-determinism.

When following [8] we use the *Seq* for an asynchronous-time model and *Par* for the synchronous one. This was demonstrated by our implementations: when running this model with *Par* the beautiful patterns emerge as reported but when following the *Seq* approach they wont and all Agents defect after a given number of generations.

5 Implementation Comparisons

5 Pages In this section we give a brief overview of implementation issues in three languages which fundamentally differ among each other. We discuss how to map the strategies to the given languages, discuss programming-language technical details and compare the languages in regard of their suitability to implement each of these strategies.

5.1 Languages

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

5.1.1 Java

This language is included as the benchmark of object-oriented (OO) imperative languages as it is extremely popular in the ABM/S community and widely used in implementing ABM/S. It comes with a comprehensive programming library, has powerful object-oriented features built in and is easy to learn although the OO concepts can be difficult to master.

5.1.2 Haskell

This language is included to put to test whether such a pure functional declarative programming language is suitable for full-blown ABM/S. 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.

5.1.3 Scala with Actors

This multi-paradigm functional language is included to test the usefulness of the *Act* strategy for implementing ABM/S. 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.

5.2 Selected Models

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 *Spatial Game* mentioned in [8]. 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 ABM/S implementation: sending messages, accessing the environment, spawning new Agents, killing existing ones, discrete and continuous model.
TODO: check for reference-models and look if all can be implemented: [9] and StupidModel 1-16

5.3 Results[1]

5.3.1 Java

TODO: create simple diagrams for java, they are very specific to the language sind doch zu spezifisch. give UML Sequence-Diagrams for better understanding of the flow of control. Although we wanted to keep this section programming-language agnostic, the Diagrams will be provided in the context of object-orientation because its semantics are widely understood and accepted.
parallelism and concurrency very easy due to elegant and powerful built-in synchronization primitives, high-performance and large number of agents possible due to aliasing and side-effects
sequential more natural in java, parallel needs to "think functional" in java concurrency and actors always difficult in java despite java provides very good synchronization primitives
sequential is able to work completely without messages and only by accesing references to neighbours but we explicitly don't want to follow this obvious way and stick to the send/receive message paradigm. we keep Agent-instance references
parallel then needs to utilize messages because would violate the parallel implementation. TODO: split

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

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

5.4 Haskell

to implement immediate message-handling in haskell would be very cumbersome: drag all agents around, could run into recursion, state-handling becomes cumbersome, =¿ leads to re-building an OO kind-of system in a pure functional language =¿ don't do that! The conclusion is that for now this is left as the single drawback, which is not appropriately implementable using Haskell. Thats really where Java shines with its mutable Objects and Side-Effects.
TODO: could we really implement synchronous message-handling in Haskell? we would need to pass always ALL agents around and return them in every call
We initially thought that Haskell would be suitable best for just implementing the Par-Strategy after implementing all the strategies in it we found out that Haskell is extremely well suited to implement all the strategies. We think this stems from the following facts that it has no implicit side-effects which reduces bugs considerably and reveals the data-flow very explicitly.
dont have objects with methods which can call between each other but we need some way of representing agents. this is done using a struct type with a behaviour function and messaging mechanisms. important: agents are not carried arround but messages are sent to a receiver identified by an id. This is also a big difference to java where don't really need this kind of abstraction due to the use of objects and their 'messaging'. messaging mechanisms have up- and downsides, elaborate on it.
extremely powerful static typesystem: in combination with side-effect free this results in the semantics of an update-strategy to be reflected in the Agent-Transformer function and the messaging-interface. This means a user of this approach can be guided by the types and can't abuse them. Thus the lesson learned here is that *if one tries to abuse the types of the agent-transformer or work around, then this is an indication that the update-strategy one has selected does not match the semantics of the model one wants to implement.* If this happens in Java, it is much more easier to work around by introducing global variables or side-effects but this is not possible in Haskell.
Thus our conclusion in using Haskell is that it is an extremely underestimated language in ABM/S which should be explored much more as we have shown that it really shines in this context and we believe that it could be pushed further even more.
it is not possible to send a message directly to an agent
note that the difference between SEQ and PAR in Haskell is in the end a 'fold' over the agents in the case of SEQ and a 'map' in the case of PAR
Also it is a must-have for STM, although it makes things more difficult in the beginning, in the end it turns out to be a blessing because one can guarantee that side-effects won't occur. We have taken care that the agents all run in side-effect free code. concurrency and actors extremely elegant possible through: STM which only possible in languages without side-effect STM: implementing concurrency is a piece-of-cake the conc-version and the act-version of the agent-implementations look EXACTLY the same BUT we lost the ability to step the simulation!!!
but still it is not suitable for big models with heterogeneous agents, there things are lacking: see further research But: still much work to do to capture large and complex models (see further research), per-

formance is a big issue but this has not been about performance (2000 Agents are enough)

5.5 Scala with Actors

direct support for actors

6 Conclusion

In this paper we presented all possible general update-strategies for an ABM/S 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 ABM/S community needs a unified terminology of speaking about update-strategies otherwise confusions arise and reproduceablility 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. Finally we can say that the usage of Scala with Actors: extremely elegant solutions possible when one is willing to sacrifice reproduceability due to non-determinism.

7 Further Research

7.1 Functional Reactive Programming

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

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

TODO: see paper-abstract

7.3 Reasoning in Haskell about the Model & Simulation

TODO: sketch ideas

7.4 Immediate Message-Handling in Haskell

This is the single main drawback of the Haskell implementation and although it only shows up in the Seq Strategy it would be of interest if there is an elegant, pure functional software-architecture which allows messages sent from Agent A to Agent B be immediately - as in: the same execution thread - handled by Agent B.

7.5 The Actor Model in ABM/S

We find that the Actor Model should get more attention in ABM/S and although we showed that the Act-Strategy implemented in Scala with Actors can implement very different kind of Models we barely scratched the surface. We hope that more research is going into this topic as we feel 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 socioeconomic systems. Documents de travail du Centre d'Economie de la Sorbonne 10076, Université Panthéon-Sorbonne (Paris 1), Centre d'Economie de la Sorbonne, 2010.

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

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

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

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

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

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

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

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

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