1st Year Report
Pure Functional Methods in Agent-Based Modelling & Simulation

Jonathan Thaler
jonathan.thaler@nottingham.ac.uk

May 15, 2017

Abstract

So far specifying Agent-Based Models and implementing them as an Agent-Based Simulation (ABS) is done using object-oriented methods like UML and object-oriented programming languages like Java. The reason for this is that until now the concept of an agent was always understood to be very close to, if not equals to - which it is not - the concept of an object. Therefore, the reasoning goes, object-oriented methods and languages should apply naturally to specify and implement agent-based simulations. In this thesis we fundamentally challenge this assumption by investigating how Agent-Based Models and Simulations (ABS) can be specified and implemented using pure functional methods and what the benefits are. We show how ABS can be implemented in the pure functional language Haskell and develop a basic underlying theoretical framework in Category-Theory and Type-Theory which allows to view and specify ABS from a new point-of-view. Having these tools at hand the major benefit using them is the potential for an unprecedented approach to validation & verification of an ABS. First: due to the declarative nature of pure functional programming in Haskell it is possible to implement an EDSL for ABS which ideally results in code which is equals to specification thus closing the gap between specification and implementation. Second: validation becomes possible by formulating hypotheses directly in code using the EDSL and QuickCheck and then verify these hypotheses. Third: Category-Theory allows to view ABS from a high-level view thus giving a different insight and allows to specify ABM in a new way.

Contents

Chapter 1

Introduction

1.1 Background

Computer simulation is the means of imitating real-world processes over time through computational means [14]. It is used for a vast number of purposes, most prominently to study effects of varying conditions when testing a real system is not feasible either because it is too dangerous or the system does not exist under such isolated circumstances [104]. Simulation has established itself as a third way of doing science and can be seen as a *generative* approach which is in opposition to the classical inductive (finding patterns in empirical data) and deductive (proving theorems) [6], [8]. The main benefit of the generative approach is that analytically intractable deductive approaches become possible through simulation. Of course one can not deduce generally valid theorems from simulations as when using deduction but often this is not necessary: validating hypotheses by interpreting the dynamics and emergent patterns of simulative experiments is often enough. Thus the generative approach can be seen as a form of empirical research and is a natural environment for studying interdisciplinary phenomena [36].

Naturally there are many different types of simulations and we focus in this thesis on Agent-based simulation and modelling (ABS) which is a method for simulating the emergent behaviour of a system by modelling and simulating the interactions of its sub-parts, called agents [103], [8], [125]. Examples for an ABS is simulating the spread of an epidemic throughout a population or simulating the dynamics of segregation within a city. Epstein [36] identifies ABS especially powerful to be in problems for analysis of *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. Central to ABS is the concept of an agent who needs to be updated in regular intervals during the simulation so it can interact with other agents and its environment. 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 known. 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 [126].

We informally assume the following about our agents [127]:

- 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 as above.
- They can interact with an environment they are situated in.

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. This explicit distinction is necessary as to not confuse our concept of *agent* with those of the field of Multi Agent Systems (MAS). ABS and MAS influence each other in the way that the basic concept of an agent is very similar but the areas of application are fundamentally different. MAS is primarily used as an engineering approach to organize large software-systems [124] where ABS is primarily used for the simulation of collective behaviour of agents to gain insight into the dynamics of a system [31].

ABS is a method and thus always applied in a very specific domain in which phenomenon are being researched which can be mapped to collective behaviour which implies that we need to select a specific domain in which we want to advance the methodology of ABS. Being a vastly diverse, and inherently interdisciplinary research domain [88] for our PhD we pick the fields of Social Sciences and Agent-Based Computational Economics (ACE). The former one is traditionally a highly interdisciplinary field [7], drawing on lots of different other areas like economics, epidemology, genetics, neurocognition, biology. It is still a young field, having emerged in the 90s but is constantly growing and gaining significance [6]. It offers the new approach of *generative* social sciences in which one tries to generate phenomenon by e.g. constructing artificial societies or introducing neurocognition to test hypotheses [38], [37]. ACE follows basically the same approach but in the field of economics to study economic processes as dynamic systems of interacting agents [112]. ACE is regarded to offer a new approach to economics, much closer to reality as compared to neo-classical economics which today forms much of the underlying theory of economy. The neo-classical approach follows a top-down approach by postulating equilibrium-properties about systems and trying to fit the agents into this framework. This generally leads to agents as being homogenous, having perfect information and acting rational [67]. On the one hand this approach allows to treat the problems analytically but on the other hand poses little resemblance to reality. A major

critique to neo-classical economics is that it is both non-constructive and uncomputable [117]. ACE on the other hand allows a constructive approach and is computable by its very nature, by following a bottom-up process in which the agents are heterogenous, have only local information at hand, act with bounded rationality and interact continuously with each other [113], [67]. This allows ACE to study economic systems which are not in equilibrium and makes it possible to investigate whether they reach equilibrium or not and under which conditions the equilibrium is reached. This shows, that the study of out-of-equilibrium models is also one of the main fields ACE, and ABS in general is suited for [36].

1.2 Problem

The challenges one faces when specifying and implementing an ABS are manifold:

- How is an agent represented?
- How do agents pro-actively act?
- How do agents interact?
- How is the environment represented?
- How can agents act on the environment?
- How to handle structural dynamism: creation and removal of agents?

Epstein & Axtell explicitly advocate object-oriented programming in [38] as "a particularly natural development environment for Sugarscape specifically and artificial societies generally." and report about 20.000 lines of code which includes GUI, graphs and plotting. They implemented their Sugarscape software in Object Pascal and C where they used the former for the Agents and the latter for low-level graphics [9]. Axelrod [6] recommends Java for experienced programmers and Visual Basic for beginners. Up until now most of ABS seems to have followed this suggestion and are implemented using programming languages which follow the object-oriented imperative paradigm. The main concept in this paradigm are objects which provide abstraction, hiding implementation details and expose an abstract interface to the user of the object who does not (and should not) make any assumptions about implementation details. So in this paradigm the program consists of an implicit, global mutable state which is spread across multiple objects.

Although object-oriented programming was invented to give programmers a better way of composing their code, strangely objects ultimately do *not* compose [17], [39]. The reason for this is that objects hide both *mutation* and *sharing through pointers or references* of object-internal data. This makes data-flow mostly implicit due to the side-effects on the mutable data which

is globally scattered across objects. To deal with the problem of composability and implicit data-flow the seminal work [43] put forward the use of *patterns* to organize objects and their interaction. TODO: add SOLID and Dependency Injection. Although a huge step in the right direction, patterns are often difficult to understand and to apply and don't solve the fundamental problem [70]. To put it short: even for experienced programmers, proper object-oriented programming *is hard* where the difficulty arises from how to split up a problem into objects and their interactions and controlling the implicit mutation of state which is spread across all objects. Still if one masters the technique of object-oriented program-design and implementation, due to the implicit global mutable state bugs due to side-effects are the daily life of a programmer as shown below. Note that this critique of object-oriented programming addresses the deficits of this paradigm as it is implemented and in use today in languages likes Java and C++. The original idea of OO invented by Alan Kay `http://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented` has much more common with the Actor Model as will be discussed in the literature-review. Another serious problem of object-oriented implementations is the blurring of the fundamental difference between agent and object - an agent is first of all a metaphor and *not* an object. In object-oriented programming this distinction is obviously lost as in such languages agents are implemented as objects which leads to the inherent problem that one automatically reasons about agents in a way as they were objects - agents have indeed become objects in this case. the single strength of todays oop is its use as a modelling language. the problem is what is going on under the hood: the sharing of mutable state. this is the real problem of mixing up the concept of object and agent

In [7] Axelrod reports the vulnerability of ABS to misunderstanding. Due to informal specifications of models and change-requests among members of a research-team bugs are very likely to be introduced. He also reported how difficult it was to reproduce the work of [5] which took the team four months which was due to inconsistencies between the original code and the published paper. The consequence is that counter-intuitive simulation results can lead to weeks of checking whether the code matches the model and is bug-free as reported in [6]. The same problem was reported in [63] which tried to reproduce the work of Gintis [45]. In his work Gintis claimed to have found a mechanism in bilateral decentralized exchange which resulted in walrasian general equilibrium without the neo-classical approach of a tatonement process through a central auctioneer. This was a major break-through for economics as the theory of walrasian general equilibrium is non-constructive as it only postulates the properties of the equilibrium [28] but does not explain the process and dynamics through which this equilibrium can be reached or constructed - Gintis seemed to have found just this process. Ionescu et al. [63] failed and were only able to solve the problem by directly contacting Gintis which provided the code - the definitive formal reference. It was found that there was a bug in the code which led to the "revolutionary" results which were seriously damaged through this error. They also reported ambiguity between the informal model description in Gintis paper and the actual implementation. This lead to a research in a functional

framework for agent-based models of exchange as described in [19] which tried to give a very formal functional specification of the model which comes very close to an implementation in Haskell. This was investigated more in-depth in the thesis by [41] who got access to Gintis code of [45]. They found that the code didn't follow good object-oriented design principles (all was public, code duplication) and - in accordance with [63] - discovered a number of bugs serious enough to invalidate the results. This reporting seems to confirm the above observations that proper object-oriented programming is hard and if not carefully done introduces bugs. The author of this text can report the same when implementing [38]. Although the work tries to be much more clearer in specifying the rules how the agents behave, when implementing them still some minor inconsistencies and ambiguities show up due to an informal specification. The fundamental problems of these reports can be subsumed under the term of verification which is the checking whether the implementation matches the specification. Informal specifications in natural language or listings of steps of behaviour will notoriously introduce inconsistencies and ambiguities which result in wrong implementations - wrong in the way that the *intended* specification does not match the *actual* implementation. To find out whether this is the case one needs to verify the model-specification against the code. Verification is a required approach for software systems which needs to meet high standards e.g. Medicine, Power Plant (TODO: cite) but has so far not been undertaken for ABS (TODO: is this really the case?, look into literature)

As ABS is almost always used for scientific research, producing often breakthrough scientific results as pointed out in [7], these ABS need to be *free of bugs*, *verified against their specification*, *validated against hypotheses* and ultimately be *reproducible*. One of the biggest challenges in ABS is the one of validation. In this process one needs to connect the results/dynamics of the simulation to initial hypotheses e.g. *are the emergent properties the ones anticipated? if it is completely different why?*. It is important to understand that we always *must have* a hypothesis regarding the outcome of the simulation, otherwise we leave the path of scientific discovery. We must admit that sometimes it is extremely hard to anticipate *emergent patterns* but still there must be *some* hypothesis regarding the dynamics of the simulation otherwise it is just guesswork.

In the concluding remarks of [7] Axelrod explicitly mentions that the ABS community should converge both on standards for testing the robustness of ABS and on its tools. However as presented above, we can draw the conclusion that there seem to be some problems the way ABS is done so far. We don't say that the current state-of-the-art is flawed, which it is not as proved by influential models which are perfectly sound, but that it always contains some inherent danger of embarrassing failure.

1.3 Motivation

The observations of the problems presented in the previous section leads us to posing fundamental directions and questions which are the basis of the motiva-

tion of our thesis.

1. **Alternative approach to object-oriented ABS** - Is there an alternative view to the established object-oriented view to ABS which does treat agents *not* like objects and does not mix the concept of agent and object? Is there an alternative to the established object-oriented implementation approach to ABS which offers composability, explicit data-flow (Hypothesis: pure functional programming, actor model, category-Theory)

2. **Verification & Validation of ABS** - Is there a way for formal specification which is still readable and does not fall back to pure mathematics? Is there a way to formally specify hypotheses about the simulation which are then checked automatically against the results? (Hypothesis: EDSL in Haskell with reasoning techniques)

Chapter 2

Literature Review

In this we present a literature-review which is driven by the motivating questions from the introduction. We present relevant sources which pose possible answers and directions of approaches to the posed questions. Based upon this information, in the next chapter we will select our directions and methods for our research, identify the gap we have to bridge with our PhD research and our objectives in achieving to close the gap.

2.1 Actor Model

TODO: this is an oo alternative

The Actor-Model, a model of concurrency, was initially conceived by Hewitt in 1973 [52] and refined later on [50], [51]. It was a major influence in designing the concept of agents and although there are important differences between actors and agents there are huge similarities thus the idea to use actors to build agent-based simulations comes quite natural. The theory was put on firm semantic grounds first through Irene Greif by defining its operational semantics [**?**] and then Will Clinger by defining denotational semantics [27]. In the seminal work of Agha [1] he developed a semantic mode, he termed *actors* which was then developed further [2] into an actor language with operational semantics which made connections to process calculi and functional programming languages (see both below).

An actor is a uniquely addressable entity which can *in response to a message*

- Send an arbitrary number (even infinite) of messages to other actors.
- Create an arbitrary number of actors.
- Define its own behaviour upon reception of the next message.

In the actor model theory there is no restriction on the order of the above actions and so an actor can do all the things above in parallel and concurrently at the same time. This property and that actors are reactive and not pro-active

is the fundamental difference between actors and agents, so an agent is *not* an actor but conceptually nearly identical and definitely much closer to an agent in comparison to an object. There have been a few attempts on implementing the actor model in real programming languages where the most notable are Erlang and Scala.

2.1.1 Erlang

The programming-model of actors [1] was the inspiration for the Erlang programming language which was created in the 80s by Eriksson [3] for developing distributed high reliability software in telecommunications. There exists very little research in using Erlang for ABS where the following papers are all what could be found on this topic.

In [116] the authors introduce reflexive action - sending messages to itself - to make the passive actors pro-active, thus rendering them in fact being agents. Although their work is rather focused on the field of MAS, it is also applicable to ABS where the main contribution of the authors is the implementation of a Belief-Desire-Intention (BDI) architecture for agents built on top of Erlang. The work of [102] emphasises the conceptual gap between the programming languages which implement ABS and the concept of an agent and propose using Erlang to close this gap. Although the paper is not very in-depth, it gives extensive code-listings for implementing Schellings Tipping Model and an agent-based representation of a neural network. In [16] a naive clone of NetLogo in the Erlang programming language is briefly discussed where each agent is represented as an Erlang process. Although the author does not go into much detail, the synchronization problems caused by the inherent asynchronous nature of the actor model become apparent when mapping ABS to the actor model. The authors of [33] and [34] focus more on using Erlang to implement MAS applications instead of ABS and propose a new agent platform called eXAT in Erlang. Besides claiming to be a more natural approach to agent-programming they mention better readability as compared to Java implementations.

2.1.2 Scala

Scala is a multi-paradigm language which also comes with an implementation of the actor-model as a library which enables to do actor-programming in the way of Erlang. It was developed in 2004 and became popular in recent years due to the increased availability of multi-core CPUs which emphasised the distributed, parallel and concurrent programming for which the actor-model is highly suited. As for Erlang, there exists even less research in using Scala & Actors for ABS.

In [69] the authors use Scala to build a massively-concurrent multi-agent system for evolutionary optimization in the domain of biology. They compared their implementation to Erlang and showed that Scala performs better due to a more efficient underlying memory-model. The paper [114] focuses explicitly on simulation using a MAS built in Scala. They compare their implementation

in Scala to a threaded version in Java and come to the finding that the Scala-version is slower. This seems to be curious as Scala with Actors should, when not slowed down by synchronization, have about the same performance. The reason for this is that their implementation does exactly this kind of synchronization by sending global time-stamps to all agents which allows them to perform the next step in their calculation thus synchronizing them all in lock-step.

2.1.3 Alan Kays Object-Oriented Programming

Alan Kay, the inventor of the OO idea had a system like the actor-model in mind when he conceived oop `http://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented`. It follows the actor model in the essence that each object has a mailbox to which other objects can send immutable messages which contain non-shareable state (e.g. no pointers, references,...). Erlang follows this approach strictly but Scala allows one to circumvent this approach by sending mutable messages and references thus violating the locality of state.

2.2 Process Calculi

TODO: this is an oo alternative

Process Calculi were initially invented for algebraically specifying concurrent systems and their interaction. By mapping concurrent computation to the field of algebra allows to reason about whether two processes are equivalent or not, whether deadlocks can occur and to formally verify correctness of concurrent systems. The main concept in process calculi is the one of independently computing processes which interact with each other via messages which act as points of synchronization between these processes. The first process calculi which were created at around the same time are Communicating Sequential Processes (CSP) [53] and Calculus of Communicating Processes (CCS) [81]. A problem of these early process calculi was that connections - the direction of message-flow - between processes were always fixed in advance. This was addressed by Milner in the π-calculus [84], [85], [82], [83] in which processes can be mobile: connections between processes can change over time. By introducing a changing network-structure of communication the π-calculus is reminiscent to agents which can also be seen as independently running processes which interact with each other through messages. Indeed, research exists which tries to specify agent-based systems in terms of the π-calculus [40], [66] and in [94] the authors model an agent-based Spanish fish market. Unfortunately no research was found on using a process calculus in the field of agent-based *simulation*.

TODO: DEVS?

2.3 (Pure) Functional Programming

TODO: this is an oo alternative

In his 1977 ACM Turing Award Lecture, John Backus, one of the giants of Computer Science and a main contributor to Fortran, an imperative programming language, fundamentally critizied imperative programming for its deep flaws and proposed a functional style of programming to overcome the limitations of imperative programming [11]. The main criticism is its use of *state-transition with complex states* and the inherent semantics of state-manipulation. In the end an imperative program consists of a number of assign-statements resulting in side-effects on global mutable state which makes reasoning about programs nearly impossible. Backus proposes the so called *applicative* computing, which he termes *functional programming* which has its foundations in the Lambda Calculus [24]. The main idea behind it is that programming follows a declarative rather than an imperative style of programming: instead of describing *how* something is computed, one describes *what* is computed. This concept abandons variables, side-effects and (global) mutable state and resorts to the simple core of function application, variable substitution and binding of the Lambda Calculus. Although possible and an important step to understand the very foundations, one does not do functional programming in the Lambda Calculus [80], as one does not do imperative programming in a Turing Machine. In our thesis we opt for Haskell as our choice for a functional programming language and deliberately ignored other functional languages [1]. The paper of [56] gives a comprehensive overview over the history of the language, how it developed and its features and is very interesting to read and get accustomed to the background of the language. A widely used introduction to programming in Haskell is [62]. The main points why we decided to go for Haskell are

- Pure, Lazy, Higher-Order and Static Typing - these are the most important points for the decision as they form the very foundation for composition, correctness, reasoning and verification.
- Real-World applications - The strength of Haskell has been proven through a vast amount of highly diverse real-world applications [2] [56] and is applicable to a number of real-world problems [93].
- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science e.g. parallelism & concurrency.
- In-house knowledge - The School of Computer Science of the University of Nottingham has a large amount of in-house knowledge in Haskell which can be put to use and leveraged in my thesis.

It seems that we are on the right track with pure functional programming in answering the questions in the motivation as it promises to solve all the issues

[1] We did a bit of research using Scala (a mixed paradigm functional language) in ABS (see Appendix B) but it is completely out-of-scope of this thesis to do an in-depth comparison of functional languages for their suitability to implement ABS.

[2] `https://wiki.haskell.org/Applications_and_libraries`

raised in these questions. We will now investigate by looking into relevant literature if this is really the case.

The main conclusion of the classical paper [58] is that *modularity* is the key to successful programming and can be achieved best using higher-order functions and lazy evaluation provided in functional languages like Haskell. The author argues that the ability to divide problems into sub-problems depends on the ability to glue the sub-problems together which depends strongly on the programming-language. He shows that laziness and higher-order functions are in combination a highly powerful glue and identifies this as the reason why functional languages are superior to structure programming. Another property of lazy evaluation is that it allows to describe infinite data-structures, which are computed as currently needed. This makes functions possible which produce an infinite stream which is consumed by another function - the decision of *how many* is decoupled from *how to*.

In the paper [120] Wadler describes Monads as the essence of functional programming (in Haskell). Originally inspired by monads from category-theory (see below) through the paper of Moggi [86], Wadler realized that monads can be used to structure functional programs [119]. A pure functional language like Haskell needs some way to perform impure (side-effects) computations otherwise it has no relevance for solving real-world problems like GUI-programming, graphics, concurrency,... . This is where monads come in, because ultimately they can be seen as a way to make effectful computations explicit [3]. In [120] Wadler shows how to factor out the error handling in a parser into monads which prevents code to be cluttered by cross-cutting concerns not relevant to the original problem. Other examples Wadler gives are the propagating of mutable state, (debugging) text-output during execution, non-deterministic choice. Further applications of monads are given in [120], [121], [122] where they are used for array updating, interpreting of a language formed by expressions in algebraic data-types, filters, parsers, exceptions, IO, emulating an imperative-style of programming. This seems to be exactly the way to go, tackling the problems mentioned in the introduction: making data-flow explicit, allowing to factor out cross-cutting concerns and encapsulate side-effects in types thus making them explicit.

The concept of monads was further generalized by Hughes by the concept of arrows [59]. The main difference between Monads and Arrows are that where monadic computations are parameterized only over their output-type, Arrows computations are parametrised both over their input- and output-type thus making Arrows more general. In [60] Hughes gives an example for the usage for Arrows in the field of circuit simulation. Streams are used to advance the simulation in discrete steps to calculate values of circuits thus the implementation is a form of *discrete event simulation* - which is in the direction we are heading already with ABS. As will be shown below, the concept of arrows is essential for Functional Reactive Programming a potential way to do ABS in pure functional

[3] This is seen as the one of the main impacts of Haskell on the mainstream programming [56]

programming.

One of the most compelling example to utilize pure functional programming is the reporting of [57] where in a prototyping contest of DARPA the Haskell prototype was by far the shortest with 85 lines of code (LoC) as compared to the C++ solution with 1105 LoC. The remarkable thing is that the Jury mistook the Haskell code as specification because its approach was to implement a small embedded domain specific language (EDSL) to solve the problem - this is a perfect proof how close an EDSL can get to a specification. When implementing an EDSL one develops and programs primitives e.g. types and functions in a host language (embed) in a way that they can be combined. The combination of these primitives then looks like a language specific to a given domain. The ease of development of EDSLs in pure functional programming is also a proof of the superior extensibility and composability of pure functional languages over OO and definitely one of its major strength. The classic paper [49] gives a wonderful way of constructing an EDSL do denotationally construct an picture reminiscent of the works of Escher. A major strength of developing an EDSL is that one can reason about and do formal verification. The testing-library QuickCheck [25], [26] defines itself an EDSL which allows to formulate a specification in the QuickCheck- EDSL and domain-EDSL and test the code against this specification - testing code happens by writing formal specifications which is the very heart of verification. It seems that in EDSL we have found a way to tackle the problem of verification and close the gap between specification and implementation at least conceptually - whether this is really possible will be subject of the research conducted in the thesis.

2.3.1 ABS

The amount of research on using the pure functional paradigm using Haskell in the field of ABS has been moderate so far. Most of the papers look into how agents can be specified using the belief-desire-intention paradigm [32], [109], [64]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [106]. It comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. [64] which discuss using functional programming for DES mention the paradigm of functional reactive programming (FRP) to be very suitable to DES. [101] and [118] present a domain-specific language for developing functional reactive agent-based simulations. This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Yampa code - a FRP library for Haskell - which they claim is also readable. It seems that FRP is a promising approach to ABS in Haskell, an important hint we will follow in the section below.

Tim Sweeney, CTO of Epic Games gave an invited talk in which he talked about programming languages in the development of game-engines and scripting of game-logic [110]. Although the fields of games and ABS seem to be very different, in the end they have also very important similarities: both are simulations which perform numerical computations and update objects in a loop either

concurrently or sequential [4]. In games these objects are called *game-objects* and in ABS they are called *agents* but they are conceptually the same thing. The two main points Sweeney made were that dependent types could solve most of the run-time failures and that parallelism is the future for performance improvement in games. He distinguishes between pure functional algorithms which can be parallelized easily in a pure functional language and updating game-objects concurrently using software transactional memory (STM).

The thesis of [16] constructs two frameworks: an agent-modelling framework and a DES framework, both written in Haskell. They put special emphasis on parallel and concurrency in their work. The author develops two programs: HLogo which is a clone of the NetLogo agent-modelling framework and HDES, a framework for discrete event simulation - where in both implementations is the very strong emphasis on parallelism.

Although probably the most important selling point of a pure functional language is its ease of parallelizing code due to lack of side-effects [97], [93], [65], [78] we don't go into this direction in our thesis and consider this just to be a by-product which luckily just falls out of the language itself [5].

2.3.2 Functional Reactive Programming

So far we have considered only quite low-level approaches to structuring and composing functional programming: higher-order functions, laziness, monads and arrows. What we need is a programming paradigm built into pure functional programming which we can leverage to implement ABS. As already mentioned above, functional reactive programming (FRP) seems to be a highly promising approach. It is rather a lucky coincidence that Henrik Nilsson, one of the major contributor to the library Yampa, an implementation of FRP, is situated at the School of Computer Science of the University of Nottingham.

FRP is a paradigm for programming hybrid systems which combine continuous and discrete components. Time is explicitly modelled: there is a continuous and synchronous time flow. There have been many attempts to implement FRP in libraries which each has its benefits and deficits. all started with fran, a domain specific language for graphics and animation and at yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa, the most recent FRP library [89], To conclude: when programming systems in Haskell and Yampa one describes the system in terms of signal functions in a declarative manner (functional programming) using the EDSL of Yampa. During execution the top level signal functions will then be evaluated and return new signal functions (transition functions) which act as continuations: "every

[4] Gregory [47] defines computer-games as *soft real-time interactive agent-based computer simulations*

[5] We did some research of implementing concurrent agents with STM as proposed by [110] and [16] in our research on programming paradigms as can be seen in Appendix B. We think that STM is probably the single major feature which is *only* possible in a pure functional language because only in a pure functional language with explicit side-effects it is possible to *compose* concurrency.

signal function in the dataflow graph returns a new continuation at every time step". "A major design goal for FRP is to free the programmer from 'presentation' details by providing the ability to think in terms of 'modeling'. It is common that an FRP program is concise enough to also serve as a specification for the problem it solves" [**?**]. [**?**] discuss the semantic framework of FRP. Very difficult to understand and full of corollaries and theorems and proofs, have to study in depth at another time.

Yampa has been used in multiple agent-based applications: [55] uses Yampa for implementing a robot-simulation, [29] implement the classical Space Invaders game using Yampa, [90] implements a Pong-clone, the thesis of [79] shows how Yampa can be used for implementing a Game-Engine, [87] implemented a 3D first-person shooter game with the style of Quake 3 in Yampa. Note that although all these applications don't focus explicitly on agents all of them inherently deal with kinds of agents which share properties of classical agents: game-entities, robots,... Other fields in which Yampa was successfully used were programming of synthesizers, network routers, computer music development and has been successfully combined with monads [96].

This leads to the conclusion that Yampa is mature, stable and suitable to be used in functional ABS. This and the reason that we have the in-house knowledge lets us focus on Yampa. Also it is out-of-scope to do a in-depth comparison of the many existing FRP libraries.

2.3.3 Dependent Types

As already pointed out by Sweeney in [110], dependent types could remove an important class of run-time errors which in the end means that using them allows to push correctness even further because type-invariants are statically checked at compile time. As correctness and verification is our major concern, dependent types seem to be attractive. The papers of [91], [21] and [20] give a good introduction of what dependent types are and how to program with them in Agda, a dependently typed pure functional programming language, closely related to Haskell. For now this approach seems to be too early to follow as we haven't yet laid the basic groundwork: an non-dependently typed pure functional implementation of ABS in Haskell.

2.3.4 Foundations in Category-Theory

With the advent of monads the interest in category-theory surged and it was discovered that many computational concepts can be expressed through category-theory [98], [107]. Because many concepts of functional programming were put on firm grounds it would be only consequential to find a category-theoretical approach to functional ABS and put its foundations on firm category-theoretical grounds as well. So far only two papers looked into category-theoretical approaches to agent-based models and simulating [15], [75] but none of them is really satisfying, making this an ideal research topic for the thesis.

2.3.5 A final word on LISP

Being the oldest functional programming language and the 2nd oldest high-level programming language ever created, at one point we considered using LISP in our research due to its immensely powerful feature of homoiconicity. The idea was to investigate if this could be made useful for ABS and bring it to a new level. We abandoned this quickly as it would have led to a total different approach. Besides, it would have definitely not solved the issues the questions raised in the introduction because of its imperative nature. Still there exists a paper [66] which implements a MAS in LISP.

2.4 Verification & Validation of ABS

In the work of [9] the authors tried to see whether the more complex Sugarscape model can be used to reproduce the results of [5]. In both models agents have a tag for cultural identification which is comprised of a string of symbols. The question was whether Sugarscape, focusing on generating a complete artifical society which incorporates many more mechanisms like trading, war, ressources can reproduce the results of [5] which only focuses on transmission of these cultural tags. Although interesting the question if two models are qualitatively equivalent is not what we want to pursue in our thesis as it requires a complete different direction of research.

TODO: write about ABS as a new tool and generative as opposed to the classical inductive and deductive sciences. major sources: TODO fully read [35] TODO fully read [36]

To do verification we need a form of formal specification which can be translated easily to the code. Being inspired by the previously mentioned work on a functional framework for agent-based models of exchange in [19] we opt for a similar direction. Having Haskell as the implementation language instead of an object-oriented one like Java allows us to build on the above proposed EDSL for ABS. Because of the declarative nature of the hypothesized EDSL it can act both as specification- and implementation-language which closes the gap between specification and implementation. This would give us a way of formally specifying the model but still in a more readable way than pure mathematics. This form of formal specification can act easily as a medium for communication between team-memberes and to the scientific audience in papers. Most important the explicit step of verification becomes obsolete as there exists no more difference between specification and implementation. The last point seems to be quite ambitious but this is a hypothesis and we will see in the course of the thesis how far we can close the gap in the end with this approach.

TODO: need to do lots of literature research still Having the EDSL from the firt two hypotheses at hand it may be possible to extend it through additional primitves which then allow to formulate hypotheses in a formal way which then can be checked automatically.

TODO: [68]

TODO: baas: emergence, hierarchies and hyperstructures TODO: [10]

TODO: Burton and Obel 1995 "The validity of computational models in organization science: from model realism to purpose of the model" TODO: Knepell 1993 "Simulation validation, a confidence assessment methodology."

TODO: http://jasss.soc.surrey.ac.uk/12/1/1.html TODO: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4419595 TODO: http://dspace.stir.ac.uk/handle/1893/3365#.WNjO1DsrKM8 TODO: http://www2.econ.iastate.edu/tesfatsi/DockingSimModels.pdf TODO: http://www2.econ.iastate.edu/tesfatsi/empvalid.htm TODO: http://jasss.soc.surrey.ac.uk/10/2/8.html TODO: https://www.openabm.org/faq/how-validate-and-calibrate-agent-based- TODO: https://link.springer.com/chapter/10.1007%2F978-3-642-01109-2_10 TODO: http://www3.nd.edu/~nom/Papers/ADS019_Xiang.pdf

TODO: http://www.cse.nd.edu/~nom/Papers/ads05_kennedy.pdf

The cooperative work of [9] gives insights into validation of computational models, in a process what they call "alignment". They try to determine if two models deal with the same phenomena. For this they tried to qualtiatively reproduce the same results of [5] in the Sugarscape model of [38]. Both models are of very different nature but try to investigate the qualitatively same phenomoenon: that of cultural processes. TODO: read

model checking and reasoning in [61]

In [25] introduce *QuickCheck*, a testing-framework which allows to specifify properties and invariants of ones functions and then test them using randomly generated test-data. This is an additional tool of model-specification and increases the power and strength of the verification process and more properties of a model can be expressed which are directly formulated in code through the EDSL of QuickCheck AND the EDSL of FrABS. Of course it also serves for testing (e.g. regression) and points out errors in the implementation e.g. wrong assumptions about input-data. The authors claim that the major advante of QuickCheckj is to formulate formal specifications which help in understanding a program. TODO: the question is whether it can be used for Validation as well.

Verification/Reasoning ist einer der größten Pluspunkte von rein funktionaler Programmierung, da durch den deklarativen Stil und das Fehlen von Sideeffects und Globalen Daten equational/algebraic/inductive Reasoning betrieben werden kann. Hier habe ich noch garnichts dazu gemacht, aber sollte mit den oben genannten Ideen sicherlich interessant werden - ein interessantes Paper von Graham Hutton (für den ich übrigens dieses Semester ein Tutor in seiner Haskell-Laborübung bin) gibt interessante Richtungen für Reasoning vor: http://dl.acm.org/citation.cfm?id=968579

[ ] deadlock: when messages need to be exchanged but mutual waiting [ ] silence: no more message exchange [ ] protocoll: ensure happens before / sequences (like necessary for 2D prisoner dilemma)

2.5 Fields of Application

2.5.1 Social Simulation

The most influential source on the generative social sciences can be regarded the work of Epstein and Axtell in [38] in which they indeed create an artificial society and connect observations in their simulation to phenomenon of real-world societies. They later added more research to this in [36] and most recently [37] in which Epstein tries to approach the generative social sciences from a neurocognitive approach.

look closer at the contents of the 3 major books: SugarScape, Generative, Agent_Zero

Agent_Zero: `http://sites.nationalacademies.org/cs/groups/dbassesite/documents/webpage/dbasse_175078.pdf`

The SugarScape model [38] is one of the most influential models of agent-based simulation in the social sciences. The book heavily promotes object-oriented programming (note that in 1996 oop was still in its infancy and not yet very well understood by the mainstream software-engineering industry). We ask how it can be done using pure functional programming paradigm and what the benefits and limits are. We hypothesize that our solution will be shorter (original reported 20.000 LOC), can make use of EDSL thus making it much more expressive, can utilize QuickCheck for a completely new dimension of model-checking and debugging and allows a very natural implementation of MetaABS (see Part III) due to its recursive and declarative nature.

TODO [54] TODO [92]

2.5.2 Agent-Based Computational Economics (ACE)

The field of economics is an immensely vast and complex one with many facets to it [22]. Equilibrium-theory is the very foundation of (micro) economics [28] and central to the way economists think and approach the dynamics of economic processes. This model requires a central *walrasian* auctioneer which has perfect information and assumes homogeneous, rationally acting agents. The theory then postulates the existence of an equilibrium under given properties but does not give a process or the dynamics how this equilibrium can be approached. This notion of equilibrium has always been criticized for being not realistic, making impossible assumptions e.g. perfect information and not being able to provide a process under which this equilibrium is reached [67]. The problem is that as soon as more realistic assumptions are made, the solutions become analytically intractable. This is where ACE comes in, as it allows to experimentally approach equilibrium-theory from a more realistic point-of-view by removing the central auctioneer and introducing agents with bounded rationality, local information and restricted interactions over networks [42].

Tesfatsion defines ACE as *[...] computational modelling of economic processes (including whole economies) as open-ended dynamic systems of interacting agents.* `http://www2.econ.iastate.edu/tesfatsi/ace.htm`. She gives a

broad overview [112] of ACE, discusses advantages and disadvantages and giving the four primary objectives of it which are:

1. Empirical understanding: why have particular global regularities evolved and persisted, despite the absence of centralized planning and control?
2. Normative understanding: how can agent-based models be used as laboratories for the discovery of good economic designs?
3. Qualitative insight and theory generation: how can economic systems be more fully understood through a systematic examination of their potential dynamical behaviors under alternatively specified initial conditions?
4. Methodological advancement: how best to provide ACE researchers with the methods and tools they need to undertake the rigorous study of economic systems through controlled computational experiments?

She introduces a model called *ACE Trading World* in which she shows how an artificial economy can be implemented without the *Walrasian Auctioneer* but just by agents and their interactions. She gives a detailed mathematical specification in the appendix of the paper which should allow others to implement the simulation. Other works which investigate ACE as a discipline and discuss its methodology are [111], [99], [13], [18].

During the reading we became particularly interested in the dynamics of bilateral decentralized bartering [6] and emerging networks. The reason for this is that the Sugarscape model [38], mentioned in the previous section on Social Simulation, implements this bilateral decentralized bartering and emerging of credit-networks and looks at the out-of-equilibrium dynamics. This allows us to bridge the gap from ACE to Social Simulation because considerable research in ACE goes into out-of-equilibrium models, which try to find processes which lead to the general equilibrium [45], [46], [4] and [19].

Although not directly subject of this research, to better understand trading and bartering, it is quite useful to have a basic understanding of *Market Microstructure* which deals how real markets work [7]. Introductory texts to market microstructure are [48], [12] and [74]. A highly interesting research using ABS for simulating the NASDAQ market was done in [30]. The authors where approached by NSDAQ to predict the switching to the decimal system which was enforced by the SEC in April 9th 2001. They implemented an ABS in Java to build relevant models and predicted most of the changes correctly. Other works

[6] The Sugarscape book [38], in footnote 14 on page 104, cites a few introductory works on bilateral bartering with incomplete information which we don't want to repeat here

[7] A topic we deliberately ignore is the one of *Market Design* which deals with problems real markets face. It is a very hot topic at the moment in economics, having received a number of Nobel-prices e.g. Alvin Roth who wrote an introduction to this topic for the non-expert [100]. In the preparation phase we did quite some reading in this field inspired by [105] and [23] on the problems caused by High-Frequency Trading (HFT). We were able to find quite a few papers which used ABS to research the benefits and downside of HFT: [123], [71], [128]. A broad overview of Market Design using Agent-Based Models is given in [77]

on using ABS in finance and stock markets are [72], [73], [108] and [95]. Another sub-field is autonomous and automated trading agents [76], [115].

Another topic I am particularly interested in is the one of networks as it was of central focus in my master-thesis TODO:cite on continuous double-auctions in networks. Although this topic is not unique to economics, it has received considerable attention in the last years due to the sub-prime mortgage crisis where contagion through networks was one of the primary reasons for its cause.

TODO: add literature on networks from my masterthesis. TODO cite Jackson Social and Economic networks, cite easley networks, crowds and markets

Concluding [8].

1. Bilateral decentralized bartering & trading
2. Out-Of-Equilibrium dynamics
3. Networks

[8] It is of very importance to note that this thesis does not attempt to develop or proof some economic theory. Rather the intention is to use ACE as a use-case to develop the tools and apply them directly to ACE to demonstrate the usefulness and benefit of the new tool.

Chapter 3

Reflecting the Literature

3.1 Conclusion

functional programming is our choice because it allows a deterministic, synchronized, actor style implementation with immutable messages and no sharing. and it allows algebraic and equational reasoning like process calculi

the single strength of todays oop is its use as a modelling language. the problem is what is going on under the hood: the sharing of mutable state. this is the real problem of mixing up the concept of object and agent

TODO: put at the end Thread of argumentation: actor model is nice but for the kind of simulation we want to do we need too much synchronization otherwise we end up non-deterministic & non-replicable. also actor-model very difficult to reason about because of non-deterministic specifications of message-transmission. what we need is a combination: deterministic, synchronized traversal of agents which are represented as a pure function: immutable messages and no exchange of aliases through which state can be mutated.

process calculi are nice for algebraic reasoning but are too cumbersome and not feasible for real, complex ABS. You do not program a large system in the lambda calculus, as you would not program a real distributed system in a process calculus [1]. they may be of use for verification & validation later on of small, critical parts of the ABS-communication which can be mapped to e.g. the pi-calculus and then apply algebraic reasoning. As emphasised in the literature-review, not research was found on using process-calculi in the field of ABS. Although we can reason that if the π-calculus can be used to specifying and reasoning about MAS then it should be possible to do so for ABS or parts of it. There exists also a connection from the actor-model to process calculi [2], which strengthen our argument.

the concept of homoiconicity of LISP seems to be very interesting and pow-

[1] it was shown by TODO: cite that the pi-calculus can encode the lambda calculus, thus it is conceptually on a very low level: too much raw power leads to chaos.

erful to apply to ABS but an agent-based model where this extremely powerful technique can be applied is lacking. Also it is clear that when using this technique verification and validation becomes immensely more difficult, if not even impossible. Thus we refrain from LISP and its homoiconicity and look for something more structured, static and not as dynamic.

This leads us to pure functional programming with FRP which combines all the benefits: local memory, messages, switching of behaviour, algebraic reasoning, static typesystem

By the literature-review it seems that all the problems of object-oriented programming (as it is done in Java and C++) mentioned in the introduction, can be solved by (pure) functional programming which abandons the concept of global state, Objects and Classes and makes data-flow explicit. This then allows to reason about correctness, termination and other properties of the program e.g. if a given function exhibits side-effects or not. Other benefits are fewer lines of code, easier maintainability and ultimately fewer bugs thus making functional programming the ideal choice for scientific computing and simulation and thus also for ACE.

3.2 Actor Model

upside: extreme huge number of agnets possible due to distributed and parallel technology downside: depends on system & hardware: scheduler, system time, systime resolution (not very nice for scientific computation), much more complicated, debugging difficult due to concurrency, no global notion of time appart from systemtime, thus always runs in real-time, but there is no global notion of time in the actor model anyway, no EDSL full of technical details, no determinism, no reasoning

Agents more a high-level concept, Actors low level, technical concurrency primitives

This makes simulations very difficult and also due to concurrency implementing a sync conversation among agents is very cumbersome. I have already experience with the Actor Model when implementing a small version of my Master-Thesis Simulation in Erlang which uses the Actor Model as well. For a continuous simulation it was actually not that bad but the problem there was that between a round-trip between 2 agents other messages could have already interfered - this was a problem when agents trade with each other, so one has to implement synchronized trading where only messages from the current agent one trades with are allowed otherwise budget constraints could be violated. Thus I think Erlang/Akka/Actor Model is better suited for distributed high-tolerance concurrent/parallel systems instead for simulations. Note: this is definitely a major point I have to argue in my thesis: why I am rejecting the actor model.

AKKA: thus my prediction is: akka/actor model is very well suited to simulations which 1. dont rely on global time 2. dont have multi-step conversations: interactions among agents which are only question-answer. TODO: find some classical simulation model which satisfies these criterias.

how can we simulate global time? how can we implement multistep conversations (by futures)?

The real problem seems to be concurrency but i feel we can simulate concurrency by synchronizing to continuous time. computations are carried out after another but because time is explicitly modelled they happen logically at the same time. these rules hold: an agent cannot be in two conversations at the same time, the agent can be in only one or none conversation at a given time t.

What if time is of no importance and only the continuous dynamics are of interest?

To put it another way: real concurrency (with threads) makes time implicit by connecting it to the real-time of the real-world, which is what one does NOT want in simulation. Maybe FRP is the way to go because it allows to explicitly model continuous and discrete time, but I have to get into FRP first to make a proper judgement about its suitability.

[**?**] describes in chapter 3.3 a naive clone of NetLogo in the Erlang programming language where each agent was represented as an Erlang process. The author claims the 1:1 mapping between agent and process to "be inherently wrong" because when recursively sending messages (e.g. A to B to A) it will deadlock as A is already awaiting Bs answer. Of course this is one of the problems when adopting Erlang/Scala with Akka/the Actor Model for implementing agents *but it is inherently short-sighted to discharge the actor-model approach just because recursive messaging leads to a deadlock.* It is not a problem of the actor-model but merely a very problem with the communication protocol which needs to be more sophisticated than [**?**] described. The hypothesis is that the communication protocol will be in fact *very highly application-specific* thus leading to non-reusable agents (across domains, they should but be re-usable within domains e.g. market-simulations) as they only understand the domain-specific protocol. This is definitely NOT a drawback but can't be solved otherwise as in the end (the content of the) communication can be understand to be the very domain of the simulation and is thus not generalizable. Of course specific patterns will show up like "multi-step handshakes" but they are again then specifically applied to the concrete domain.

3.3 Pure Functional Programming

why pure functional programming? what are its strengths? why does it overcome the problems? what are possible problems when doing it pure functional instead of oop?

3.3.1 Strengths

- Pure - explicit about effects through monads
- Composability through higher-order functions and lazyness
- EDSL by its declarative style

- Reasoning through EDSL and lack of implicit side-effects
- Static Type System

3.3.2 Weaknesses

Here we give an overview of the weaknesses and problems of pure, lazy functional programming in Haskell.

Space-Leaks The main issue in a lazy functional programming language is the difficulty of predicting space behaviour, which is very hard even for experienced programmers [56]. The problem arises from the fact, that Haskell abstracts away from evaluation order and object lifetimes. Programmers have no way to determine which data-structures live for how long - indeed they don't want and should not be bothered to think about these details as this would violate the whole concept behind pure lazyness [56].

Debugging Due to the lazy evaluation and non-imperative programming style it becomes apparent that debugging needs to be approached completely different than in imperative programming where one can freely set breakpoints to statements and inspect data. This is not possible in Haskell as there are no imperative statements and the data may have not been evaluated yet due to the unpredictable evaluation order as mentioned already in the space-leak problem.

Performance Due to the lack of side-effects and aliasing, efficient in-order updates of memory is not as easily possible as in imperative languages like C thus real-time applications like Games which have a big global mutable state run much slower compared to its imperative implementations. TODO: check if these references really support my argument [87], [79] - The works on game-programming in Yampa mention a similar problem (FRP-section).

Monad composition Monads do not compose in a nice, modular way and this issue is still open research [56]. TODO: is this still the case?

Chapter 4

Aims and Objectives

In this chapter we derive the hypotheses to the motivating questions from the introduction based upon our literature-review and present our aims and objectives for our PhD. We present research-questions and hypotheses.

4.1 Deriving the research-direction

TODO: select the direction: Category-Theory and not Actor Model, Haskell with FRP and not Scala&Actors/Erlang. need a thorough explanation reasoning

emphasize my own research on scala with actors, experiments with erlang and prototyping with haskell.

As becomes evident from the literature-review we advocate pure functional programming in Haskell and its category-theoretic foundations as a solution to the questions posed. The usage of pure functional programming in ABS is also a strong motivation by itself as it hasn't been researched yet and deserves a thorough treatment on its own. Surprisingly there exist hardly any attempts on implementing ABS in pure functional programming as will become clear in the literature-review. Maybe this can also be seen as a hint that ABS lacks a level of deductive formalism which we hope to repair with our thesis.

So put short the motivation is a twofold direction, referring to each other in a circular way. First, pure functional programming has not been researched for implementing and specifying ABS so far. Second, the current state-of-the-art seems to be susceptible to flaws and bugs due to the lack of powerful verification. Combining both issues forms the very basic motivation of our thesis: use pure functional programming and its underlying theoretical framework to develop new methods for specifying, implementing, verifying and validating ABS to create simulations which are more reliable, reproducible and communicatable.

4.2 Identifying the Gap

- Functional programming in this area exists but only scratches the surface and focus only on implementing agent-behaviour frameworks like BDI. An in-depth treatise of Agent-Based Modelling and implementing an Agent-Based Simulation in a pure functional language has so far never been attempted.

- There basically exists no approach to Agent-Based Modelling & Simulation in terms of Category-Theory and Type-Theory

- Verification is an issue in ABS as they are very often described in natural language and supplemented with a few formulas. This leads to implementation-errors, e.g. Gintis Bartering-Paper, and results become hard to reproduce. Such errors become a threatening problem when simulation-results are used in decision making e.g. economics, policy-making, ...

- Validation is basically an untouched topic in ABS: models are formulated, a few hypotheses are formulated, the model is implemented and run, then the results are checked against the hypotheses. What the field of ABS needs is an in-depth discussion on how to rigorously validate a model. Validation is of course only as strong as the verification part: if the implementation is wrong anyway then we can not rely on anything (from false comes nothing)

- developing a category- & type-theoretical view on Agent-Based Modelling & Simulation which will -¿ 1. give a deeper insight into the structure of agents, agent-models and agent-based simulation -¿ 2. serves as the basis for the pure functional implementation -¿ 3. serves as a high-level specification tool for agent-models

- implementing a library called FrABS based upon the FRP paradigm which allows to specify Agent-Based Models in an EDSL and run them

- Verification: closing the gap between specification and implementation through the category- & type-theoretical view and the EDSL

- Validation: formalizing hypotheses and reasoning about dynamics and expected outcomes of the simulation

Define 5 general research questions for each Research-Context

- 2 related to FP
- 1 related to integration of FP to ABM/S
- 2 related to ABM/S

4.3 Aims

The pure functional programming language Haskell offers all the required features. Together with the concept of functional reactive programming (FRP) we will show that ABS becomes very well possible to be implemented in a functional language. The declarative nature of Haskell allows us to implement an embedded domain-specific language (EDSL) for ABS based on the FRP paradigm.

we propose a new tool for implementing, testing, verifying and validating ABS: the one of pure functional programming and its underlying theory which

we together will term *pure functional methods*. We claim that these methods are the answer to the questions posed above and will reduce or even elimiate the danger of failure considerably as outlined in the hypotheses. Of course our motivation is not only to improve verification, validation and reproduceability in ABS but also the one of discovery. So far pure functional programming was not really investigated in the context of ABS as will be shown in the literature-review so this thesis is also a proof-of-concept, an investigation how ABS can be done in a pure functional language.

The central aspect of this thesis is centred around the main question of *How Agent-Based Simulation can be done using pure functional programming and what the benefits and disadvantages are.*. So far functional programming has not got much attention in the field of ABS and implementations always focus on the object-oriented approach. We claim, based upon the research of the first year that functional programming is very well suited for ABS and that it offers methods which are not directly possible and only very difficult to achieve with object-oriented programming.

We claim that to build large and complex agent-based simulations in functional programming is possible using the functional reactive programming (FRP) paradigm. We applied FRP to implementing ABS and developed a library in Haskell called FrABS. We implemented the quite complex model SugarScape from social simulation using FrABS and proofed by that, that applying FRP to ABS enables ABS to happen in pure functional programming.

After having shown how agent-based simulation can be done in functional programming we claim that the major benefit of using it enabled a new way of *verification & validation* in agent-based simulation.

Due to the declarative nature of pure functional programming it is an established method of implementing an EDSL to solve a given problem in a specific domain. We followed this approach in FrABS and developed an EDSL for ABS in pure functional programming. Our intention was to develop an EDSL which can be used both as specification- and implementation-language. We show this by specifying all the rules of SugarScape in our EDSL.

Due to the lack of implicit side-effects and the recursive nature of pure functional programming we claim that it is natural to apply it to a novel method we came up with: MetaABS, which allows recursive simulation.

Finally having such an EDSL at hand this will allow us to reason about the programs. This will be applied to specify and reason about the dynamics and emergent properties of decentralized bilateral trading and bartering in agent-based computational economics (ACE) and social simulations like SugarScape.

Disadvantages - although the lack of side-effects is also a benefit, it is also a weakness as all data needs to be passed in and out explicitly - indirection due to the lack of objects & method calls. - When not to use it: - if you are not familiar with functional programming - when you can solve your problem without programming in a Tool like NetLogo, AnyLogic,... - when you don't need to reason about your program

Functional approach to Agent-Based Modelling & Simulation Because we left the path of OO and want to develop a completely different method we have

fundamentally two problems to solve in our functional method: 1. Specifying the Agent-Based Model (ABM): Category-Theory, Type-Theory, EDSL: all this clearly overlaps with the implementation-aspect because the theory behind pure functional programming in Haskell is exactly this. This is a very strong indication that functional programming may be able to really close the gap between specification and implementation in ABS. 2. Implementing the ABM into an Agent-Based Simulation (ABS): building on FRP paradigm

We show that the implicit assumption that an Agent is *about equal to* an object is not correct and leads to many implicit assumptions in OO implementations of an ABS. When implementing ABS in Haskell these implicit assumptions become explicit and challenge the fundamental assumptions about ABS and Agents. We present these implicit assumption in an explicit way by approaching it through programming, type-theory and category-theory to further deepen the concepts and methods in the field of Agent-Based Modelling & Simulation.

TODO: is it really valid to bind the sending of messages to the advancing of time? Downside: we cannot have method-calls as in OO, which allows agents to communicate with each other without time advancing.

Expected benefits 1. By mapping the concepts of ABS to Category-Theory and Type-Theory we gain a deeper understanding of the deeper structure of Agents, Agent-Models and Agent-Simulations. 2. The declarative nature of pure functional programming will allow to close the gap between specification and code by designing an EDSL for ABS in Haskell building on the previously derived abstractions in Category-Theory and Type-Theory. The abstractions and the EDSL implementation will then serve as a specification tool and at the same time code. 3. The pure functional nature together with the EDSL and abstractions in Category- & Type-Theory allow for a new level of formal verification & validation using a combination of mathematical proofs in Category- & Type-Theory, algebraic reasoning in the EDSL and model-checking using Unit-Tests and QuickCheck. The expectation is that this allows us to formally specify hypotheses about expected outcomes about the dynamics (or emergent patterns) of our simulations which then can be verified.

4.4 Objectives

4.4.1 Functional reactive ABS in Haskell

implement full-fledged pure functional ABS library on top of Haskell and FRP (Yampa): FrABS -¿ includes a very basic view on ABS from the perspective of category-theory

4.4.2 Category Theory view on ABS

4.4.3 Case Study in Verification

Sugarscapes decentralized bilateral trading

4.4.4 Case Study in Validation

Agent_Zero network extension

Semantics for FrABS and our EDSL to reason about the results: are they reasonable? do they match the theory? if yes why? if not why not? - Can we define semantics for the EDSL to do reasoning about ABS in general? - How can we reason about ABS in general in pure functional programming? - dynamics - emergent properties - deadlocks - silence (no messages/agent-agent communication and interaction) - define semantics of FrABS based on semantics of FRP and Actors - what is emergence in ABS and how can we reason about it? - identify emergent properties: equilibrium, behaviour on macroscale not defined on micro, chaos,... - can we anticipate emergent properties / dynamics just by looking at the code and reason about it? - can emergence in ABS be formalized? - hypothesis: it may be possible through functional programming because of its dual nature of declarative EDSL which awakens to a process during computation - what is the relation between emergence and computation? we need change over time (=computation) for emergence

- Can we reason about the dynamics and equilibria of agent-based models of decentralized bilateral trading & bartering?

Chapter 5

Work To Date

TODO: Describe the research work carried out during this stage of the PhD and the outcomes. A literature review must be included. Then, as appropriate according to the PhD project, this section can also include theoretical and/or experimental methods, presentation and discussion of results, etc. In the case that papers have been submitted or published within the year of the review, this section can be shorter and focused on discussing the outcomes from those papers within the wider context of the PhD programme of study (papers to be included in the appendix).

We began the search for oo alternatives initially with experimenting in Erlang and then ended up in haskell

5.0.1 Papers Submitted

Update-Strategies in ABS SSC 2017, Deadline in 31st March, 25-29th September 2017, `http://www.sim2017.com/`

TODO: attach as appendix

A foundational paper

5.0.2 Paper Drafts

Programming Paradigms and ABS SSC 2018, Deadline in March 2018?

TODO: attach as appendix

In this work I investigated the suitability of three fundamentally different programming pardigms to implement an ABS. The paradigms I looked at was object-oriented using Java, pure functional using Haskell and multi-paradigm functional using Scala with the Actors library. It is important to note that at this point I didn't use FRP as underlying paradigm in Haskell TODO: would this have changed my final conclusion on its suitability?

STM: the really unique thing which is ONLY possible in pure functional programming is composition of concurrency. TODO: cite Tim Sweeny Actors in Scala

5.0.2.1 Papers in Progress

Recursive ABS dont pursue recursive ABS further as separate paper: it could take a whole PhD by itself. proof-of-concept is ok but not groundbreaking. maybe some really groundbreaking idea will come up in a few months, then can work on it again. incorporate it maybe in FrABS paper. definitely incorporate it in final thesis

basic work and prototype within Schelling Segregation is running. The main finding for now is that it does not increase the convergence speed to equilibrium but can lead to extreme volatility of dynamics although the system seems to be near to complete equilibrium. In the case of a 10x10 field it was observed that although the system was nearly in its steady state - all but one agents were satisfied - the move of a single agent caused the system to become completele unstable and depart from its near-equilibrium state to a highly desequilibrium state.

Give each Agent the ability to run the simulation locally from its point of view do anticipate its actions and change them in the future thus introducing a meta-level in the simulation, from which the method derives its name.

- TODO: i have only the idea but am lacking a theory or hypothesis for its use

- meta need a kind of decision error measure to distinguish between various meta-simulations. also we need a mechanism to sample the decision space =¿ it can be considered to be an optimization technique.

Problems

- Definition of a recursive, declarative description of the Model.
- Perfect information about other agents is not realistic and runs counter to agent-based simulation (especially in social sciences) thus an Agent needs to be able to have local, noisy representations of the other agents.
- Local representation of other agents could be captured by Hidden Markov Models: observe what other agents do but have hidden interpretation of their internal state - these internal state-representations can be different between the local and the global version whereas the agent learns to represent the global version as best as possible locally.
- Infinite regress is theoretically possible but not on computers, we need to terminate at some point

Interpretation: It can be regarded as a Model of Free Will in ABS, which allows learning in an ABS environment in a new way - look on the section of interpretation. Application: hypothesis: allows to model social and psychological phenomena like free will. Mostly in social sciences, maybe also in economics. Investigate SugarScape, PrisonersDilemma and ACE Trading World

TODO: question: what is the meaning of an entity running simulations? it strongly depends on the context: in ACE it may be search for optimization behaviour, in Social Simulation it may be interpreted as a kind of free will

Research Questions

1. How does deep regression influence the dynamics of a system? Hypothesis: TODO
2. How do the dynamics of a system change when using perfect information or learning local information? Hypothesis: TODO
3. Is a hidden markov model suitable for the local learning? Hypothesis: TODO
4. How can MetaABS best be implemented? Hypothesis: implementing a MetaABS EDSL in a pure functional language like Haskell, should be best suited due to its inherent recursive, declarative nature, which should allow a direct mapping of features of this paradigm to the specification of the meta-model

- functional programming perfect. standard toolkits (anylogic, netlogo, repast) are not capable of doing this - extend my existing EDSL for functional reactive agent-based simulation & modelling (FrABS/M) with recursive functionality

Related Research: TODO: [44] cite paper of recursive simulation: [ ] military simulation, [ ] not explicitly abs, [ ] implemented in c++, [ ] deterministic models seem to benefit significantly from using recursions of the simulation for the decision making process. when using stochastic models this benefit seems to be lost

Functional Reactive ABS TODO: put into separate chapter as this is the fundamental approach put forward in this thesis TFP 2017, Deadline 5th May 2017, 19-21th June 2017, `https://www.cs.kent.ac.uk/events/tfp17/index.html`

5.0.2.2 Software

TODO: make a better, more detailed list of what prototyping i did

Lots of prototyping: Heroes & Cowards, SIRS & Schelling Segregation in Java, Haskell and Scala Parallelism and Concurrency in Haskell

Lots of learning & prototyping for Haskell: IO, STM, Pure Functional, FRP

FrABS: SugarScape Model as use-case no.1. TODO: available on github

5.0.3 Talks

presenting the ideas of my Update-Strategies paper at the IMA - seminar day
presenting my FrABS ideas to the FP-Lab Group at the FPLunch

Chapter 6

Functional Reactive ABS (FrABS)

6.1 General principles

idea: can we implement a message between two agents through events? thus two states: waiting for messages, processing messages. BUT: then sending a message *will take some time*

NOTE: it is important to make a difference about whether the simulation will dynamically *add* or *remove* agents during execution. If this is not the case, a simple par-switch is possible to run ALL agent SF in parallel. If dynamically changes to the agent-population should be part of the simulation, then the dpSwitch or dpSwitchB should be used. Also it should be possible to start/stop agents: if they are inactive then they should have no running SF because would use up resources. Inactive means: doing nothing, also not awaiting something/"doing nothing in the sense that DOING something which is nothing - the best criteria to decide if an agent can be set inactive is when the event which decides if the agents SF should be started comes from outside e.g. if the agent is just statically "living" but not changing and then another agent will "ignite" the "living" agent then this is a clear criterion for being static without a running SF.

NOTE: the route-function will be used to distribute "messages" to the agents when they are communicating with each other

TODO: need a mechanism to address agents: if agent A wants to send a message to agent B and agent B wants to react by answering with a message to agent A then they must have a mechanism to address each other

TODO: design general input/output data-structures

TODO: design general agent SF

Wormholes in FRP?

6.2 Yampa

TODO: why Yampa? There are lots of other FRP-libraries for Haskell. Reason: in-house knowledge (Nilsson, Perez), start with *some* FRP-library to get familiar with the concept and see if FRP is applicable to ABS. TODO: short overview over other FRP-libraries but leave a in-depth evaluation for further-research out of the scope of the PhD as Yampa seems to be suitable. One exception: the extension of Yampa to Dunai to be able to do FRP in Monads, something which will be definitely useful for a better and clearer structuring of the implementation. TODO: Push vs. Pull

TODO: describe FRP

TODO: 1st year report Ivan: "FPR tries to shift the direction of data-flow, from message passing onto data dependency. This helps reason about what things are over time, as opposed to how changes propagate". QUESTION: Message-passing is an essential concept in ABS, thus is then FRP still the right way to do ABS or DO WE HAVE TO LOOK AT MESSAGE PASSING IN A DIFFERENT WAY IN FRP, TO VIEW AND MODEL IT AS DATA-DEPENDENCY? HOW CAN THIS BE DONE? BUT: agent-relations in interactions are NEVER FIXED and always completely dynamic, forming a network. The question is: is there a mechanism in which we have explicit data-dependency but which is dynamic like message-passing but does not try to fake method-calls? maybe the conversations come very close

time- and space-leak: when a time-dependent computation falls behind the current time. TODO: give reason why and how this is solved through Yampa. Yampa solves this by not allowing signals as first-class values but only allowing signal functions which are signal transformers which can be viewed as a function that maps signals to signals. A signal function is of type SF which is abstract, thus it is not possible to build arbitrary signal functions. Yampa provides primitive signal functions to define more complex ones and utilizes arrows [**?**] to structure them where Yampa itself is built upon the arrows: SF is an instance of the Arrow class.

Fran, Frob and FAL made a significant distinction between continuous values and discrete signals. Yampas distinction between them is not as great. Yampas signal-functions can return an Event which makes them then to a signal-stream - the event is then similar to the Maybe type of Haskell: if the event does not signal then it is NoEvent but if it Signals it is Event with the given data. Thus the signal function always outputs something and thus care must be taken that the frequency of events should not exceed the sampling rate of the system (sampling the continuous time-flow). TODO: why? what happens if events occur more often than the sampling interval? will they disappear or will the show up every time?

switches allow to change behaviour of signal functions when an event occurs. there are multiple types of switches: immediate or delayed, once-only and recurring - all of them can be combined thus making 4 types. It is important to note that time starts with 0 and does not continue the global time when a

switch occurs. TODO: why was this decided?

The first implementations of FRP (Fran) implemented FRP with synchronized stream processors which was also followed by [**?**]. Yampa is but using continuations inspired by Fudgets. In the stream processors approach "signals are represented as time-stamped streams, and signal functions are just functions from streams to streams", where "the Stream type can be implemented directly as (lazy) list in Haskell...":

```
type Time = Double
type SP a b = Stream a -> Stream b
newtype SF a b = SF (SP (Time, a) b)
```

Continuations on the other hand allow to freeze program-state e.g. through closures and partial applications in functions which can be continued later. This requires an indirection in the Signal-Functions which is introduced in Yampa in the following manner.

```
type DTime = Double

data SF a b =
        SF { sfTF :: DTime -> a -> (SF a b, b)
```

The implementer of Yampa call a signal function in this implementation a *transition function*. It takes the amount of time which has passed since the previous time step and the durrent input signal (a). It returns a *continuation* of type SF a b determining the behaviour of the signal function on the next step (note that exactly this is the place where how one can introduce stateful functions like integral: one just returns a new function which encloses inputs from the previous time-step) and an *output sample* of the current time-step.

When visualizing a simulation one has in fact two flows of time: the one of the user-interface which always follows real-time flow, and the one of the simulation which could be sped up or slowed down. Thus it is important to note that if I/O of the user-interface (rendering, user-input) occurs within the simulations time-frame then the user-interfaces real-time flow becomes the limiting factor. Yampa provides the function embedSync which allows to embed a signal function within another one which is then run at a given ratio of the outer SF. This allows to give the simulation its own time-flow which is independent of the user-interface.

Chapter 7

Future Work Plan

TODO: A future work plan that is consistent with the progress to date, stating clearly the research question(s) to be addressed during the next year of the PhD.

TODO: gantt chart!

7.0.1 TODOs

out of this i will build the gantt chart for the next 12 months+

7.0.1.1 Category Theory

develop category theory behind FrABS: look into monads, arrows

category theory foundations (monads, arrows)

7.0.1.2 Implementation and Software-Engineering

use monadic or arrowized programming for structuribg the siftware implement schelling segregation in recursiveABS and report results

FrABS: SugarScape 1st prototype: pure-functional implemented, no category-theory/type-theory applied 2nd prototype: category-theory/type-theory applied: clean monadic / arrowized programming applied

Agent_Zero 1st prototype: implemented the book, based upon FrABS Extension: implement networks and study the effects there. use this as the case-study for verification and validation

7.0.1.3 Verification and Validation

look into QuickCheck to test and verificate FrABS

7.0.1.4 Papers

paper 2: recursive ABS - don't pursue this : SSC 2018, Deadline in March 2018? paper 3: FrABS - Towards pure functional programming in ABS: TFP

2017, Deadline 5th May 2017, 19-21th June 2017, https://www.cs.kent.ac.uk/events/tfp17/index.html

paper 4: Towards category theory in ABS: ? paper 5: verification and validation in ABS with pure functional programming: AAMS 2018, Deadline in November 2018, http://www.aamas2017.org/

7.0.1.5 Reading

read "Writing For Computer Science" read "Agent_Zero" read "category theory for the sciences"

7.0.2 Concept of an Agent

an agent is not an object but when implementing ABS in oo then it is tempting to treat an agent like that. when implementing it in a pure functional language like haskell, this temptation cannot arise which creates a different view on agents.

- what do i want to have achieved in the first year? -¿ FrABS -¿ 1st conference paper published -¿ research questions -¿ milestones -¿ clear idea what i want to do and where to go

- Paradigm shift in ABS!

- Focus on Papers + Milestones -¿ find out which conferences I want to go and when the deadlines are

- organize as agile sprints

- Conferences -¿ Multi-Agent Systems, good for verification and validation: AAMS, Deadline in November -¿ Social Simulation, good for new methods in simulation e.g. MetaBS: SSC, Deadline in March -¿ functional, for presenting functional stuff but not too advanced: TFP -¿ TODO some functional, good for FrABS & EDSL: ?

7.0.3 Milestones

1st Year Viva

Deadline: end of July

Submitted Paper on Recursive ABS

SSC 2018, Deadline in March 2018?,

Submitted Paper on FrABS

TFP 2017, Deadline 5th May 2017, 19-21th June 2017, https://www.cs.kent.ac.uk/events/tfp17/index.html

Submitted Paper on Validation & Verification

AAMS 2018, Deadline in November 2018, http://www.aamas2017.org/

Completed PhD

Deadline September 2019

7.0.4 Time-Line

TODO: add Gantt-Chart from june 2017 on detailed overview of activity also project gantt-chart back to what i have done so far

The whole PhD lasts for 3 years, 36 Months, from October 2016 to September 2019 and thus I will structure it according to 3 years where each year will be a major milestone - which is also intended by the Computer School.

7.0.5 ToDo

1. Paper: agents' vs. agent's
2. FRP & ABS = FrABS: meeting with Henrik
3. FrABS: work on it
4. Agent_Zero genauer anschauen: winter simulation papers, unterschied zu sugarscape?
5. Refine project-plan

7.0.6 1st Year: Groundwork

In this year I will learn basics and develop and research the methodology I will use for the main work in the 2nd year. The year will be guided by the principal question of "How can Agent-Based Simulation be done using pure functional programming?".

Important mile-stones:

31st March 2017	Finished and submit Paper
June (Mid) 2017	Finished writing 1st year report
End of June / Begin of July 2017	Oral annual report

7.0.6.1 Major Activities

- Develop FrABS library: March 2017 - May 2017
- Refine FrABS and publish as library on Hackage: July 2017 - September 2017
- Write FrABS paper: July 2017 - September 2017
- Study decentralized bartering: July 2017 - September 2017

7.0.6.2 March 2017

- experiment with MetaABS (also implement it in the EDSL) to see if it is making any sense to follow this road
- Programming: bring ABS to FRP in Yampa: FrABS, implement SugarScape
- 31st March: Finalize and submit 'Art of Iterating'-Paper to SSC 2017
- Formulate Research Questions
- Draft Project Plan

7.0.6.3 April 2017

- 9th - 13th: Midland Graduate School in Leicester
- Programming: bring ABS to FRP in Yampa: FrABS, implement SugarScape
- Refine Research Questions
- Refine Project Plan

7.0.6.4 May 2017

- Start writing 1st year report
- Programming: bring ABS to FRP in Yampa: FrABS, implement SugarScape
- Finalize Research Questions
- Finalize Project Plan

7.0.6.5 June 2017

- 4th - 18th: 2 weeks holiday on Amrum (planned last year already)
- Finalize 1st year report
- Prepare for 1st Year oral exam
- End (or beginning of July) 1st Year oral exam

7.0.6.6 July 2017

- Start writing on FrABS Paper: show how the rules of SugarScape can be formalized in FrABS =¿ specification equals code
- Study decentralized bilateral trading/bartering
- Refine FrABS library

7.0.6.7 August 2017

- Work on FrABS Paper: Formalize the Rules in the EDSL of FrABS
- Study decentralized bilateral trading/bartering
- Refine FrABS library

7.0.6.8 September 2017

- Finalize FrABS Paper
- Study decentralized bilateral trading/bartering
- Refine FrABS library: put on hackage

7.0.6.9 October 2017

2nd year starts

7.0.7 2nd Year: Main Work

Applying 1st year results, methods and experiences to work in the Research-Questions. The second year is investigating the question of "What are the benefits of doing ABS in pure functional programming?".

Important mile-stones:

October	Finished FrABS Paper
?	Submit FrABS Paper
?	Finished and submit MetaABS Paper

7.0.7.1 Major Points for research

- EDSL which is both specification- and implementation- language
- MetaABS which allows to do recursive simulation
- Reasoning about dynamics and emergent properties of ABS

7.0.7.2 October 2017

Start of 2nd year

7.0.7.3 December 2017

- 22nd December 2017 - 7th January 2018: 2 weeks xmas holidays

7.0.7.4 January 2018

- 22nd December 2017 - 7th January 2018: 2 weeks xmas holidays

7.0.7.5 April 2018

1 week holiday

7.0.7.6 August 2018

2 weeks holidays

7.0.7.7 October 2018

3rd year starts

7.0.8 3rd Year: Finalizing, Publishing & Writing

The third year serves to refine, finish and publish the research of the 2nd year (ideally a journal paper) and then to write the final thesis. To have a bit of distraction and to prevent myself to become too locked in in writing on the thesis I will also work on my optional philosophical paper and hope to at least finish them and maybe publish them - at least I want to present them to 2-3 audiences (e.g. FP Lunch) to test the reaction.

Important mile-stones:

30th September 2019	End of official PhD

7.0.8.1 October 2018

3rd year starts

7.0.8.2 October 2018 - December 2018

Finalize research of 2nd year

7.0.8.3 22nd December 2018 - 6th January 2019

2 weeks xmas holidays

7.0.8.4 January 2019 - March 2019

Publish journal paper of 2nd year, define structure of PhD Thesis

7.0.8.5 April 2019

1 week holiday

7.0.8.6 April 2019 - August 2019

Writing thesis

7.0.8.7 September 2019

Submitting final thesis, 2 weeks of holidays

7.0.8.8 October 2019

End of official PhD

7.0.9 4th Year: Pending Period

It is very hard to finish ALL within the 3rd year and it is very likely that I will enter the 4th year - pending period. I plan on spending in this period not more than till Christmas as I have no funding in this time and I want to finish within time.

Important mile-stones:

1st October 2019	Start of pending period
30th September 2020	End of pending period

7.0.9.1 October

- Prepare for viva

7.0.9.2 November

- Viva

7.0.9.3 December

- Refine Thesis (incorporate minor changes)

Chapter 8

Conclusions

TODO: Provide a succinct account of the conclusions from the report, stating clearly the research questions that have been identified during this stage of the PhD and the progress so far towards addressing those questions.

It is of most importance to stress that we don't condemn the current state-of-the-art approach of object-oriented specification and implementation to ABS. The strength of object-oriented programming is surely that it can be seen as *programming as modelling* and thus will be always an attractive approach to ABS. Also we are realists and know that there are more points to consider when selecting a set of methods for developing software for an ABS than robustness, verification and validation. Almost always the popularity of an existing language and which languages the implementer knows is the driving force behind which methods and languages to choose. This means that ABS will continue to be implemented in object-oriented programming languages and many perfectly well functioning models will be created by it in the future. Although they all suffer from the same issues mentioned in the introduction this doesn't matter as they are not of central importance to most of them. Nonetheless we think our work is still essential and necessary as it may start a slow paradigm-shift and opens up the minds of the ABS community to a more functional and formal way of approaching and implementing agent-based models and simulations and recognizing the benefits one gets automatically from it by doing so.

Bibliography

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

[2] Agha, G. A., Mason, I. A., Smith, S. F., and Talcott, C. L. A Foundation for Actor Computation. *J. Funct. Program. 7*, 1 (Jan. 1997), 1–72.

[3] Armstrong, J. Erlang. *Commun. ACM 53*, 9 (Sept. 2010), 68–75.

[4] Arthur, W. B. Out-of-Equilibrium Economics and Agent-Based Modeling. Handbook of Computational Economics, Elsevier, 2006.

[5] Axelrod, R. The Convergence and Stability of Cultures: Local Convergence and Global Polarization. Working Paper, Santa Fe Institute, Mar. 1995.

[6] Axelrod, R. Advancing the Art of Simulation in the Social Sciences. In *Simulating Social Phenomena*. Springer, Berlin, Heidelberg, 1997, pp. 21–40. DOI: 10.1007/978-3-662-03366-1_2.

[7] Axelrod, R. Chapter 33 Agent-based Modeling as a Bridge Between Disciplines. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1565–1584. DOI: 10.1016/S1574-0021(05)02033-2.

[8] Axelrod, R., and Tesfatsion, L. A Guide for Newcomers to Agent-Based Modeling in the Social Sciences. Staff General Research Papers Archive, Iowa State University, Department of Economics, Jan. 2006.

[9] Axtell, R., Axelrod, R., Epstein, J. M., and Cohen, M. D. Aligning simulation models: A case study and results. *Computational & Mathematical Organization Theory 1*, 2 (Feb. 1996), 123–141.

[10] Baas, N. A., and Emmeche, C. On Emergence and Explanation. Working Paper, Santa Fe Institute, Feb. 1997.

[11] Backus, J. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[12] Baker, H. K., Kiymaz, H., Alan, N. S., Bildik, R., and Schwartz, R. Market Microstructure in Emerging and Developed Markets. *Business Faculty Book Gallery* (Jan. 2013).

[13] Ballot, G., Mandel, A., and Vignes, A. Agent-based modeling and economic theory: where do we stand? *Journal of Economic Interaction and Coordination 10*, 2 (2015), 199–220.

[14] Banks, J., Carson, J. S., Nelson, B. L., and Nicol, D. M. *Discrete-Event System Simulation: Pearson New International Edition*. Pearson Higher Ed, Aug. 2013. Google-Books-ID: JiWpBwAAQBAJ.

[15] Beheshti, R., and Sukthankar, G. Analyzing Agent-Based Models Using Category Theory. In *Proceedings of the 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 02* (Washington, DC, USA, 2013), WI-IAT '13, IEEE Computer Society, pp. 280–286.

[16] Bezirgiannis, N. *Improving Performance of Simulation Software Using Haskell's Concurrency & Parallelism*. PhD thesis, Utrecht University - Dept. of Information and Computing Sciences, 2013.

[17] Bill. What is the meaning of "doesn't compose"?, July 2017.

[18] Blume, L., Easley, D., Kleinberg, J., Kleinberg, R., and Tardos, E. Introduction to computer science and economic theory. *Journal of Economic Theory 156* (Mar. 2015), 1–13.

[19] Botta, N., Mandel, A., Ionescu, C., Hofmann, M., Lincke, D., Schupp, S., and Jaeger, C. A functional framework for agent-based models of exchange. *Applied Mathematics and Computation 218*, 8 (Dec. 2011), 4025–4040.

[20] Bove, A., and Dybjer, P. Dependent Types at Work. In *Language Engineering and Rigorous Software Development*, A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, Eds., no. 5520 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 57–99. DOI: 10.1007/978-3-642-03153-3_2.

[21] Bove, A., Dybjer, P., and Norell, U. A Brief Overview of Agda — A Functional Language with Dependent Types. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2009), TPHOLs '09, Springer-Verlag, pp. 73–78.

[22] Bowles, S., Edwards, R., and Roosevelt, F. *Understanding Capitalism: Competition, Command, and Change*, 3 edition ed. Oxford University Press, New York, Mar. 2005.

[23] Budish, E., Cramton, P., and Shim, J. Editor's Choice The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response. *The Quarterly Journal of Economics 130*, 4 (2015), 1547–1621.

[24] Church, A. *The Calculi of Lambda-conversion*. Princeton University Press, 1941. Google-Books-ID: KCOuGztKVgcC.

[25] Claessen, K., and Hughes, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 268–279.

[26] Claessen, K., and Hughes, J. Testing Monadic Code with QuickCheck. *SIGPLAN Not. 37*, 12 (Dec. 2002), 47–59.

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

[28] Colell, A. M. *Microeconomic Theory*. Oxford University Press, 1995. Google-Books-ID: dFS2AQAACAAJ.

[29] Courtney, A., Nilsson, H., and Peterson, J. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (New York, NY, USA, 2003), Haskell '03, ACM, pp. 7–18.

[30] Darley, V., and Outkin, A. V. *Nasdaq Market Simulation: Insights on a Major Market from the Science of Complex Adaptive Systems*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2007.

[31] 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.

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

[33] 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.

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

[35] Epstein, J. M. Chapter 34 Remarks on the Foundations of Agent-Based Generative Social Science. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1585–1604. DOI: 10.1016/S1574-0021(05)02034-4.

[36] Epstein, J. M. *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton University Press, Jan. 2012. Google-Books-ID: 6jPiuMbKKJ4C.

[37] Epstein, J. M. *Agent_Zero: Toward Neurocognitive Foundations for Generative Social Science*. Princeton University Press, Feb. 2014. Google-Books-ID: VJEpAgAAQBAJ.

[38] Epstein, J. M., and Axtell, R. *Growing Artificial Societies: Social Science from the Bottom Up*. The Brookings Institution, Washington, DC, USA, 1996.

[39] Erkki Lindpere. Why the debate on Object-Oriented vs Functional Programming is all about composition, Dec. 2013.

[40] Esterline, A. C., and Rorie, T. Using the pi-Calculus to Model Multiagent Systems. In *Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers* (London, UK, UK, 2001), FAABS '00, Springer-Verlag, pp. 164–179.

[41] Evensen, P., and Märdin, M. *An Extensible and Scalable Agent-Based Simulation of Barter Economics*. Master, Chalmers University of Technology, Göteborg, 2010.

[42] Farmer, J. D., and Foley, D. The economy needs agent-based modelling. *Nature 460*, 7256 (Aug. 2009), 685–686.

[43] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition ed. Addison-Wesley Professional, Oct. 1994.

[44] Gilmer, Jr., J. B., and Sullivan, F. J. Recursive Simulation to Aid Models of Decision Making. In *Proceedings of the 32Nd Conference on Winter Simulation* (San Diego, CA, USA, 2000), WSC '00, Society for Computer Simulation International, pp. 958–963.

[45] Gintis, H. The Emergence of a Price System from Decentralized Bilateral Exchange. *Contributions in Theoretical Economics 6*, 1 (2006), 1–15.

[46] Gintis, H. The Dynamics of General Equilibrium. SSRN Scholarly Paper ID 1017117, Social Science Research Network, Rochester, NY, Sept. 2007.

[47] Gregory, J. *Game Engine Architecture, Third Edition*. Taylor & Francis, Mar. 2018.

[48] Harris, L. *Trading and Exchanges: Market Microstructure for Practitioners*. Oxford University Press, 2003. Google-Books-ID: Rd9hDRR1Yx4C.

[49] Henderson, P. Functional Geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (New York, NY, USA, 1982), LFP '82, ACM, pp. 179–187.

[50] Hewitt, C. What Is Commitment? Physical, Organizational, and Social (Revised). In *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, P. Noriega, J. Vázquez-Salceda, G. Boella, O. Boissier, V. Dignum, N. Fornara, and E. Matson, Eds., no. 4386 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 293–307. DOI: 10.1007/978-3-540-74459-7_19.

[51] Hewitt, C. Actor Model of Computation: Scalable Robust Information Systems. *arXiv:1008.1459 [cs]* (Aug. 2010). arXiv: 1008.1459.

[52] 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.

[53] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985. Google-Books-ID: tpZLQgAACAAJ.

[54] 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.

[55] Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, J. Jeuring and S. L. P. Jones, Eds., no. 2638 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 159–187. DOI: 10.1007/978-3-540-44833-4_6.

[56] Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 2007), HOPL III, ACM, pp. 12–1–12–55.

[57] Hudak, P., and Jones, M. Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity. Research Report YALEU/DCS/RR-1049, Department of Computer Science, Yale University, New Haven, CT, Oct. 1994.

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

[59] Hughes, J. Generalising Monads to Arrows. *Sci. Comput. Program. 37*, 1-3 (May 2000), 67–111.

[60] Hughes, J. Programming with Arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming* (Berlin, Heidelberg, 2005), AFP'04, Springer-Verlag, pp. 73–129.

[61] Hutton, G. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program. 9*, 4 (July 1999), 355–372.

[62] Hutton, G. *Programming in Haskell*. Cambridge University Press, Aug. 2016. Google-Books-ID: 1xHPDAAAQBAJ.

[63] Ionescu, C., and Jansson, P. Dependently-Typed Programming in Scientific Computing. In *Implementation and Application of Functional Languages* (Aug. 2012), R. Hinze, Ed., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 140–156. DOI: 10.1007/978-3-642-41582-1_9.

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

[65] Jones, S. P., and Singh, S. A Tutorial on Parallel and Concurrent Programming in Haskell. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (Berlin, Heidelberg, 2009), AFP'08, Springer-Verlag, pp. 267–305.

[66] Kawabe, Y., Mano, K., and Kogure, K. The Nepi2programming System: A pi-Calculus-Based Approach to Agent-Based Programming. In *Formal Approaches to Agent-Based Systems* (Apr. 2000), Springer, Berlin, Heidelberg, pp. 90–102.

[67] Kirman, A. *Complex Economics: Individual and Collective Rationality*. Routledge, London ; New York, NY, July 2010.

[68] Klügl, F., and Davidsson, P. AMASON: Abstract Meta-model for Agent-Based SimulatiON. In *Multiagent System Technologies: 11th German Conference, MATES 2013, Koblenz, Germany, September 16-20, 2013. Proceedings*, M. Klusch, M. Thimm, and M. Paprzycki, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 101–114. DOI: 10.1007/978-3-642-40776-5_11.

[69] Krzywicki, D., Turek, W., Byrski, A., and Kisiel-Dorohinicki, M. Massively concurrent agent-based evolutionary computing. *Journal of Computational Science 11* (Nov. 2015), 153–162.

[70] Lawrence Krubner. Object Oriented Programming is an expensive disaster which must end, July 2014.

[71] Leal, S. J., Napoletano, M., Roventini, A., and Fagiolo, G. Rock around the clock: An agent-based model of low- and high-frequency trading. *Journal of Evolutionary Economics 26*, 1 (Mar. 2016), 49–76.

[72] LeBaron, B. Agent-based computational finance: Suggested readings and early research. *Journal of Economic Dynamics and Control 24*, 5–7 (June 2000), 679–702.

[73] Lebaron, B. Building the santa fe artificial stock market. Working Paper, Graduate. In *School of International Economics and Finance, Brandeis* (2002), pp. 1117–1147.

[74] Lehalle, C.-A., and Laruelle, S. Market Microstructure in Practice. World Scientific Books, World Scientific Publishing Co. Pte. Ltd., 2013.

[75] Lloyd, K. A Category-Theoretic Approach to Agent-based Modeling and Simulation. *Swarmfest 2010 Santa Fe* (2010).

[76] MacKie-Mason, J. K., and Wellman, M. P. Chapter 28 Automated Markets and Trading Agents. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1381–1431. DOI: 10.1016/S1574-0021(05)02028-9.

[77] Marks, R. Chapter 27 Market Design Using Agent-Based Models. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1339–1380. DOI: 10.1016/S1574-0021(05)02027-7.

[78] Marlow, S. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013. Google-Books-ID: k0W6AQAACAAJ.

[79] Meisinger, G. *Game-Engine-Architektur mit funktional-reaktiver Programmierung in Haskell/Yampa*. Master, Fachhochschule Oberösterreich - Fakultät für Informatik, Kommunikation und Medien (Campus Hagenberg), Austria, 2010.

[80] Michaelson, G. *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation, 2011. Google-Books-ID: gKvwPtvsSjsC.

[81] Milner, R. *A Calculus of Communicating Systems*. Springer-Verlag, 1980. Google-Books-ID: 7L1PAQAAIAAJ.

[82] Milner, R. Elements of Interaction: Turing Award Lecture. *Commun. ACM 36*, 1 (Jan. 1993), 78–89.

[83] Milner, R. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, May 1999. Google-Books-ID: k2tfQgAACAAJ.

[84] Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes, I. *Information and Computation 100*, 1 (Sept. 1992), 1–40.

[85] Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes, II. *Information and Computation 100*, 1 (Sept. 1992), 41–77.

[86] Moggi, E. Computational Lambda-calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (Piscataway, NJ, USA, 1989), IEEE Press, pp. 14–23.

[87] Mun Hon, C. *Functional Programming and 3D Games*. PhD thesis, University of New South Wales, Sydney, Australia, 2005.

[88] Niazi, M., and Hussain, A. Agent-based Computing from Multi-agent Systems to Agent-based Models: A Visual Survey. *Scientometrics 89*, 2 (Nov. 2011), 479–499.

[89] Nilsson, H., Courtney, A., and Peterson, J. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (New York, NY, USA, 2002), Haskell '02, ACM, pp. 51–64.

[90] Nilsson, H., and Perez, I. Declarative Game Programming: Distilled Tutorial. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, 2014), PPDP '14, ACM, pp. 159–160.

[91] Norell, U. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (New York, NY, USA, 2009), TLDI '09, ACM, pp. 1–2.

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

[93] O'Sullivan, B., Goerzen, J., and Stewart, D. *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[94] Padget, J., and Bradford, R. A pi-calculus Model of a Spanish Fish Market - Preliminary Report -. In *Agent Mediated Electronic Commerce* (May 1998), Springer, Berlin, Heidelberg, pp. 166–188.

[95] Panayi, E., Harman, M., and Wetherilt, A. Agent-Based Modelling of Stock Markets Using Existing Order Book Data. In *Multi-Agent-Based Simulation XIII* (June 2012), Springer, Berlin, Heidelberg, pp. 101–114.

[96] Perez, I., Bärenz, M., and Nilsson, H. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell* (New York, NY, USA, 2016), Haskell 2016, ACM, pp. 33–44.

[97] Peyton Jones, S., Gordon, A., and Finne, S. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1996), POPL '96, ACM, pp. 295–308.

[98] Pierce, B. C. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, MA, USA, 1991.

[99] Richiardi, M. Agent-based Computational Economics. A Short Introduction. LABORatorio R. Revelli Working Papers Series 69, LABORatorio R. Revelli, Centre for Employment Studies, 2007.

[100] Roth, A. *Who Gets What - And Why: The Hidden World of Matchmaking and Market Design*. HarperCollins UK, June 2015. Google-Books-ID: gV_GBAAAQBAJ.

[101] Schneider, O., Dutchyn, C., and Osgood, N. Towards Frabjous: A Two-level System for Functional Reactive Agent-based Epidemic Simulation. In *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium* (New York, NY, USA, 2012), IHI '12, ACM, pp. 785–790.

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

[103] Siebers, P.-O., and Aickelin, U. Introduction to Multi-Agent Simulation. *arXiv:0803.3905 [cs]* (Mar. 2008). arXiv: 0803.3905.

[104] Sokolowski, J. A., and Banks, C. M. *Principles of Modeling and Simulation: A Multidisciplinary Approach*. Wiley, Feb. 2009. Google-Books-ID: wOOikQEACAAJ.

[105] Sornette, D., and von der Becke, S. Crashes and high frequency trading. Government Office for Science, Aug. 2011.

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

[107] Spivak, D. I. *Category Theory for the Sciences*, 1 ed. Mit Press Ltd, Cambridge, Massachusetts, Nov. 2014.

[108] Streltchenko, O., Yesha, Y., and Finin, T. Multi-Agent Simulation of Financial Markets. In *Formal Modelling in Electronic Commerce*, P. S. O. Kimbrough and P. D. J. Wu, Eds., International Handbooks on Information Systems. Springer Berlin Heidelberg, 2005, pp. 393–419. DOI: 10.1007/3-540-26989-4_15.

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

[110] Sweeney, T. The Next Mainstream Programming Language: A Game Developer's Perspective. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2006), POPL '06, ACM, pp. 269–269.

[111] Tesfatsion, L. Agent-Based Computational Economics. Computational Economics, EconWPA, Aug. 2002.

[112] Tesfatsion, L. Agent-Based Computational Economics: A Constructive Approach to Economic Theory. Handbook of Computational Economics, Elsevier, 2006.

[113] Tesfatsion, L. Modeling Economic Systems as Locally-Constructive Sequential Games. *Economics Working Papers* (Apr. 2017).

[114] Todd, A. B., Keller, A. K., Lewis, M. C., and Kelly, M. G. *Multi-agent System Simulation in Scala: An Evaluation of Actors for Parallel Simulation.*

[115] Toulis, P., Kehagias, D., and Mitkas, P. A. Mertacor: A Successful Autonomous Trading Agent. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (New York, NY, USA, 2006), AAMAS '06, ACM, pp. 1191–1198.

[116] 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.

[117] Velupillai, K. V. The unreasonable ineffectiveness of mathematics in economics. *Cambridge Journal of Economics 29*, 6 (Nov. 2005), 849–872.

[118] Vendrov, I., Dutchyn, C., and Osgood, N. D. Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling. In *Social Computing, Behavioral-Cultural Modeling and Prediction*, W. G. Kennedy, N. Agarwal, and S. J. Yang, Eds., no. 8393 in Lecture Notes in Computer Science. Springer International Publishing, Apr. 2014, pp. 385–392. DOI: 10.1007/978-3-319-05579-4_47.

[119] Wadler, P. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (New York, NY, USA, 1990), LFP '90, ACM, pp. 61–78.

[120] Wadler, P. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1992), POPL '92, ACM, pp. 1–14.

[121] Wadler, P. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text* (London, UK, UK, 1995), Springer-Verlag, pp. 24–52.

[122] Wadler, P. How to Declare an Imperative. *ACM Comput. Surv. 29*, 3 (Sept. 1997), 240–263.

[123] Wah, E., and Wellman, M. P. Latency Arbitrage, Market Fragmentation, and Efficiency: A Two-market Model. In *Proceedings of the Fourteenth ACM Conference on Electronic Commerce* (New York, NY, USA, 2013), EC '13, ACM, pp. 855–872.

[124] Weiss, G. *Multiagent Systems*. MIT Press, Mar. 2013. Google-Books-ID: WY36AQAAQBAJ.

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

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

[127] Wooldridge, M., and Jennings, N. R. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review 10* (1995), 115–152.

[128] Yim, K., Oh, G., and Kim, S. The Effect of Fast Traders in Continuous Double Auction Market. SSRN Scholarly Paper ID 2702014, Social Science Research Network, Rochester, NY, Dec. 2015.

Appendices

Appendix A

Training Courses

- Computer Science PGR Introductory Seminar 5 Dates
- Tradition of Critique Lecture series, Monday 29th September - Monday 8th December (18:00 - 20:00)
- Graduate School:
  - Nature of the doctorate and the supervision process, 15th November 2016 (9:30 - 12:00)
  - Presentation skills for researchers (all disciplines), 27th Jan 2017 (9:30 - 15:30)
  - Planning your research, 20th Feb 2017 (9:30 - 13:00)
  - Getting into the habit of writing, 23th Feb 2017 (9:30 - 12:30)
- Midland Graduate School 2017 from 9-13 April in Leicester. Attended courses on Denotational Semantics, Naïve Type Theory and Testing with Theorem Provers.

Appendix B

Update-Strategies

TODO: write short explanation of this paper: tries to develop foundational technical vocabulary for ABS.

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

Jonathan Thaler
jonathan.thaler@nottingham.ac.uk
School of Computer Science, University of Nottingham

Peer-Olaf Siebers
peer-olaf.siebers@nottingham.ac.uk
School of Computer Science, University of Nottingham

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 as simulation results can vary vastly across different update-strategies. This awareness, we claim, is still underdeveloped in the majority of the field of ABS. In this paper we propose a new terminology to classify update strategies and then identify different strategies using this terminology. This will allow implementers and researchers in this field to use a general terminology, removing ambiguities when discussing ABS and their models. We will give results of simulating a discrete and a continuous game using our update-strategies and show that in the case of the discrete game only one specific strategy seems to be able to produce its emergent patterns whereas the pattern of the continuous game seems to be robust under varying update-strategies.

Keywords: Agent-Based Simulation, Parallelism, Concurrency, Emergence

1. Introduction

Agent-based simulation (ABS) is a method for simulating the emergent behaviour of a system by modelling and simulating the interactions of its sub-parts, called agents. Examples for an ABS is simulating the spread of an epidemic throughout a population or simulating the dynamics of segregation within a city. Central to ABS is the concept of an agent who needs to be updated in regular intervals during the simulation so it can interact with other agents and its environment. In this paper we are looking at two different kind of simulations to show the differences update-strategies can make in ABS and support 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 outcome is a terminology to communicate in a unified way about this very important matter, so researchers and implementers can talk about it in a common way, enabling better discussions, same understanding and better reproducibility and continuity in research. The two simulations 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. The contribution of this paper is: Identifying general properties of ABS, deriving update-strategies from these properties and establishing a general terminology for talking about these update-strategies.

2. Background

In this section we define our 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. 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 known. 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 (Wooldridge, 2009). 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 as 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 a continuous 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 illusion of its own thread-of-execution by sharing the global thread 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 multi-processing 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: immediate or queued. 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 (Nowak & May, 1992). In the prisoners dilemma one assumes that two persons are locked up in a prison and can choose to cooperate with each other or to defect by betraying the other one. Looking at a game-theoretic approach there are two options for each player which makes four possible outcomes. Each outcome is associated with a different payoff in the prisoner-dilemma. If both players cooperate both receive payoff R; if one player defects and the other cooperates the defector receives payoff T and the cooperator payoff S; if both defect both receive payoff P where $T > R > P > S$. The dilemma is that the safest strategy for an individual is to defect but the best payoff is only achieved when both cooperate. In the version of (Nowak & May, 1992) NxN agents are arranged on a 2D-grid where every agent has 8 neighbours except at the edges. 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 and sums up the payoff. After the payoff sum is calculated the agent changes its role to the role of the agent with the highest payoff within its

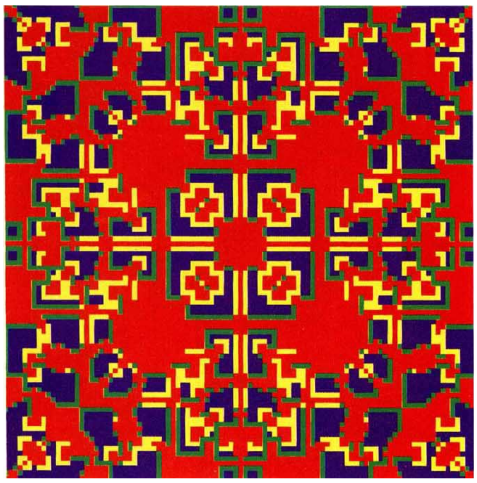


Figure 1: Patterns formed by playing the *Prisoners Dilemma* game on a 99x99 grid with $1.8 < b < 2$ after 217 steps with all agents being cooperators except one defector at the center. Blue are cooperators, red are defectors, yellow are cooperators which were defectors in the previous step, green are defectors which were cooperators in the previous step. Picture taken from (Nowak & May, 1992).

neighbourhood (including itself). The authors attribute the following payoffs: 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 (Huberman & Glance, 1993) the authors show 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 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. The authors differentiated between the two strategies but their description still lacks precision and detail, something we will provide in this paper. Although they published their work in the area on general computing, it has 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 of them. As will be shown later, the patterns emerge indeed only when selecting a specific update-strategy.

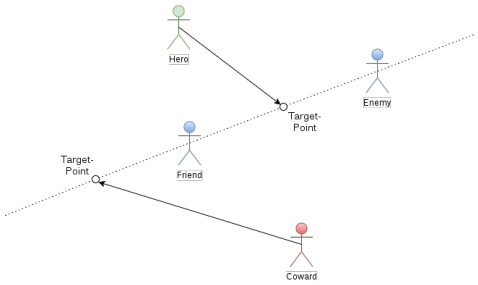


Figure 2: A conceptual diagram of the *Heroes & Cowards* game. Hero (green) and coward (red) have the same agents as friend and enemy but act different: the hero tries to move in between the friend and enemy whereas the coward tries to hide behind its friend.

2.3. A continuous game: Heroes & Cowards

As an example for a continuous game we use the *Heroes & Cowards* game introduced by (Wilensky & Rand, 2015). 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 as seen in Figure 2. 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.

2.4. Related Research

Besides (Nowak & May, 1992) and (Huberman & Glance, 1993) which both discuss asynchronous and synchronous updates, multiple other works mention these kind of updates but the meaning is different in each. Asynchronous updates in the context of cellular automata was defined by (Bersini & Detours, 1994) as picking a cell at random and updating it and syn-

chronous as all cells updating at the same time and report different dynamics when switching between the two. The authors also raise the question which of both is correct and most faithful to reality as in an ideal solution both should deliver the similar spatio-temporal dynamics. They conclude that developers of simulations should pay attention to the fact that the dynamics and results are sensible to the updating procedures. Asynchronous vs. synchronous updates are mentioned in the book of (Wilensky & Rand, 2015) where they define asynchronous updates as having the property that changes made by an agent are seen immediately by the others whereas in synchronous updating the changes are only visible in the next tick. They also look into the notion of sequential vs. parallel actions and identify as sequential when only one agent acts at a time and parallel when agents act truly parallel, independent from each other. The same argumentation is followed by (Railsback & Grimm, 2011) where they discuss the importance of order of execution of the agents and describe asynchronous and synchronous updating. Yet another definition of synchronous and asynchronous updates is given in (Page, 1997). They define asynchronous updating as updating agents sequentially one after another and synchronous updating as updating all agents at virtually the same time. They go further and discuss also random updates where the order of the agent-sequence is shuffled before updating all agents. They also introduce incentive based asynchronous updating where the agent which gains the most from the update is updated first, thus introducing an ordering on the sequence which is sorted by the benefit each agent gains from its update. They also compare the differences synchronous, random-asynchronous and incentive-asynchronous updating has on dynamics and come to the conclusion that the order of updating the agents has an impact on the dynamics and should be considered with great care when implementing a simulation. Asynchronous and synchronous time-models are mentioned in (Dawson, Siebers, & Vu, 2014) where the authors 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 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. A different interpretation of synchronous and asynchronous time-models is given in (Yuxuan, 2016). 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. A different approach was taken in (Botta, Mandel, & Ionescu, 2010) where they sketch a minimal ABS implementation in Haskell. Their research applies primarily to 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. A very different approach to updating and iterating agents in ABS than to mechanisms used in existing software like AnyLogic or NetLogo was given in (Lysenko, D'souza, & Rahmani, 2008) where the authors mapped ABS on GPUs. 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 can be utilized to roughly map their approach onto our terminology.

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 standard 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 for speaking about update-strategies in ABS in general. We will discuss all details programming-language agnostic and for each strategy we give a short description, the list of all properties and discuss their semantics, variations and implications selecting update-strategies for a model. A summary of all update-strategies and their properties is given in Table 1.

3.1. ABS Properties

We identified the following properties of agent-based simulations which are necessary to derive and define the differences between the 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*, 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 others? Note that it seems to add 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 is against our definition of agents which allows them to have 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 agents $a_{n>i}$ which are updated after agent a_i see the changes by agent a_i or not? If yes, we refer to *In-Iteration* visibility, to *Post-Iteration* otherwise.

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*, otherwise *deterministic*. It is important to distinguish between *external* and *internal* sources of non-determinism. The former 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 an ABS non-deterministic. The latter, coming from random-number generators, can be controlled using the same starting-seed leading to repeatability and deemed deterministic in this context.

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 infinite-loops.

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. 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 them 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 in a way 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 information, 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 and 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. It is important to note that in this strategy a reply to a message will not be delivered in the current but in the next global time-step. This is in contrast to the immediate message-delivery of the *sequential* strategy where within a global time-step agents can have in fact an arbitrary number of messages exchanged.

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 act 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.

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 (Hewitt, Bishop, & Steiger, 1973) for which (Greif, 1975) and (Clinger, 1981) 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.

3.3. ABS Toolskits

There exist a lot of tools for modelling and running ABS. We investigated the abilities of two of them to capture our update-strategies and give an overview of our findings in this section.

3.3.1. NetLogo. NetLogo is probably the most popular ABS toolkit around as it comes with a modelling language which is very close to natural language and very easy to learn for non-computer scientists. It follows a strictly single-threaded computing approach when running a single model, so we can rule out both the *concurrent* and *actor strategy* as both require separate threads of execution. The tool has no built-in concept of messages and it is built on global synchronization which is happening through advancing the global time by the 'tick' command. It falls into the responsibility of the model-implementer to iterate over all agents and let them perform actions on themselves and on others. This allows for very flexible updating of agents which also allows to implement the *parallel strategy*. A NetLogo model which implements the prisoners dilemma game synchronous and asynchronous to reproduce the findings of (Huberman & Glance, 1993) can be found in chapter 5.4 of (Jansen, 2012).

3.3.2. AnyLogic. AnyLogic follows a rather different approach than NetLogo and is regarded as a multi-method simulation tool as it allows to do system dynamics, discrete event simulation and agent-based simulation at the same time where all three methods can interact with each other. For ABS it provides the modeller with a high-level view on agents and does not provide the ability to iterate over all agents - this is done by AnyLogic itself and the modeller can customize the behaviour of an agent either by modelling diagrams or programming in Java. As NetLogo, AnyLogic runs a model using a single thread thus the *concurrent* and *actor strategy* are not feasible in AnyLogic. A feature this toolkit provides is communication between agents using messages and it supports both queued and immediate messaging. AnyLogic does not provide a mechanism to directly implement the *parallel strategy* because all changes are seen immediately by the other agents but using queued messaging we think that the *parallel strategy* can be emulated nevertheless.

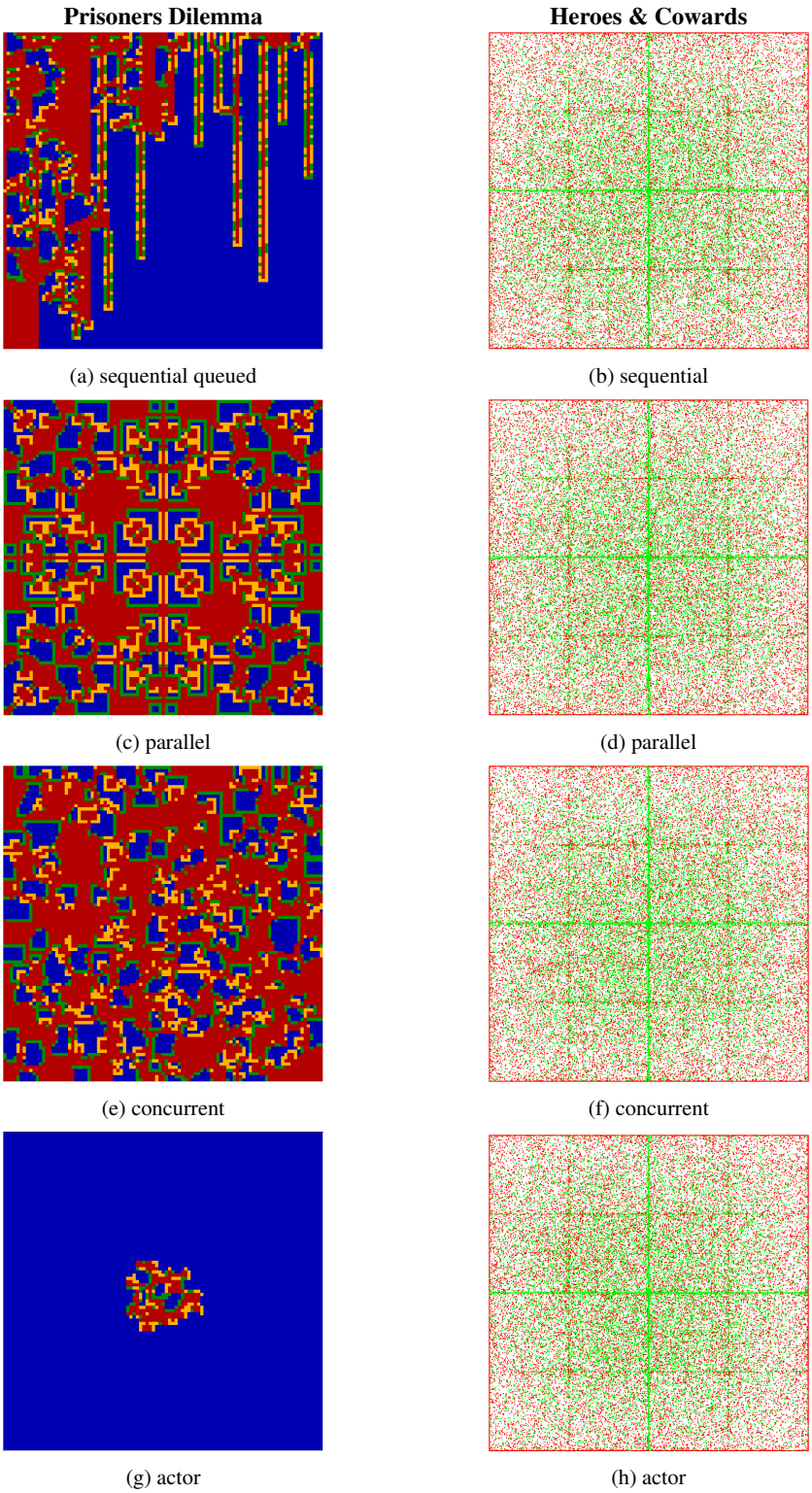
3.3.3. Summary. To conclude, the most natural and common update-strategy in these toolkits is the *sequential strategy* which is not very surprising. The primary target are mostly agent-based modellers which are non-computer scientists so the toolkits also try to be as simple as possible and multi-threading and concurrency would introduce lots of additional complications for modellers to worry about. So the general consensus is to refrain from multi-threading and concurrency as it is obviously harder to develop, debug and introduces non-repeatability in the case of concurrency and to stick with the *sequential strategy*. The *parallel strategy* is not supported *directly* by any of them but can be implemented using various mechanisms like queued message passing and custom iteration over the agents.

4. 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 shown below. We implemented the simulations for all strategies except the *actor strategy* in Java and the simulations for the *actor strategy* in Haskell and in Scala with the Actor-Library.

4.1. Prisoners Dilemma

The agent-based model Our agent-based model of this game works as follows: at the start of the simulation each agent sends its state to all its neighbours which allows to incrementally calculate the local payoff. If all neighbours' states have been received then the agent will send its local payoff to all neighbours which allows to compare all payoffs in its neighbourhood and calculate the best. When all neighbours' local payoffs have been received the agent will adopt the role of the highest payoff and sends its new state to all its neighbours, creating a circle.



(a) sequential queued

(b) sequential

(c) parallel

(d) parallel

(e) concurrent

(f) concurrent

(g) actor

(h) actor

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

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

Care must be taken to ensure that the update-strategies are comparable because when implementing a model in an update-strategy it is necessary to both map the model to the strategy and try to stick to the same specification - if the implementation of the model differs fundamentally across the update-strategies it is not possible to compare the solutions. So we put great emphasis and care keeping all four implementations of the model the same just with a different update-strategy running behind the scenes which guarantees comparability.

Results The results as seen in the left column of Figure 3 were created with the same configuration as reported in (Nowak & May, 1992). When comparing the pictures with the one from the reference seen in Figure 1 the only update-strategy which is able to reproduce the matching result is the *parallel strategy* - all the others clearly fail to reproduce the pattern. From this we can tell that only the *parallel strategy* is suitable to simulate this model.

To reproduce the pattern of Figure 1 the simulation needs to be split into two global steps which must happen after each other: first calculating the sum of all payoffs for every agent and then selecting the role of the highest payoff within the neighbourhood. This two-step approach results in the need for twice as many steps to arrive at the matching pattern when using *queued* messaging as is the case in the *parallel, concurrent* and *actor strategy*.

For the *sequential strategy* one must further differentiate between *immediate* and *queued* messaging. We presented the results using the *queued* version, which has the same implementation of the model as the others. When one is accepting to change the implementation slightly, then the *immediate* version is able to arrive at the pattern after 217 steps with a slightly different model-implementation: because immediate messaging transfers the thread of control to the receiving agent that agent can reply within this same step. This implies that we can calculate a full game-round (both steps) within one global time-step by a slight change in the model-implementation: an agent sends its current state in every time-step to all its neighbours.

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 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. Also agents will send messages to itself which will be processed in the same time-step. 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. Although agents don't change unless all their neighbours have answered, this does not guarantee a synchronized update of all agents because every agent has a different neighbourhood which is reflexive but not transitive. If agent a is agent's b neighbour and agent c is agent's b neighbour this does not imply that agent c is agent's a neighbour as well. This allows the spreading of changes throughout the neighbourhood, resulting in a breaking down of the pattern. This is not the case in the *parallel strategy* where 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 introduced by concurrency.

4.2. Heroes & Cowards

The agent-based model Our agent-based model of this game works as follows: in each time-step an agent asks its friend and enemy for their positions which will answer with a corresponding message containing their current positions. The agent will have its own local information about the position of its friend and enemy and will calculate its move in every step based on this local information.

Results The results as seen in the right column of Figure 3 were created with 100.000 agents where 25% of them are heroes running for 500 steps. 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 agents in the tens of thousands - we went for 100.000 because then the patterns are really prominent. The patterns form because the heroes try to stay halfway between their friend and enemy: with this high number of cowards it is very likely that heroes end up with two cowards - the cowards will push towards the border as they try to escape, leaving the hero in between. 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.

5. Conclusion and future research

In this paper we have presented general properties of ABS, derived four general update-strategies and discussed their implications. By doing this we proposed a unified terminology which allows to speak about update-strategies in a common and unified way, something that the ABS community is currently lacking. We hope our classification and terminology will help the community to better understand the details necessary to consider implementing an agent-based simulations. 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 showed that 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 breakdown 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 in future research.

References

Bersini, H., & Detours, V. (1994). Asynchrony induces stability in cellular automata based models. In *In Proceedings of Artificial Life IV* (pp. 382–387). MIT Press.

Botta, N., Mandel, A., & Ionescu, C. (2010). *Time in discrete agent-based models of socio-economic systems* (Documents de travail du Centre d'Economie de la Sorbonne No. 10076). Université Panthéon-Sorbonne (Paris 1), Centre d'Economie de la Sorbonne.

Clinger, W. D. (1981). *Foundations of Actor Semantics* (Tech. Rep.). Cambridge, MA, USA: Massachusetts Institute of Technology.

Dawson, D., Siebers, P. O., & Vu, T. M. (2014, September). 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* (pp. 1453–1460). doi: 10.15439/2014F335

Greif, I. (1975). *Semantics of communicating parallel processes* (Tech. Rep.). Cambridge, MA, USA: Massachusetts Institute of Technology.

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

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

Jansen, M. (2012). *Introduction to Agent-Based Modeling.* Retrieved from `https://www.openabm.org/book/introduction-agent-based-modeling`

Lysenko, M., D'souza, R., & Rahmani, K. (2008). *A Framework for Megascale Agent Based Model Simulations on the GPU*.

Nowak, M. A., & May, R. M. (1992, October). Evolutionary games and spatial chaos. *Nature*, *359*(6398), 826–829. doi: 10.1038/359826a0

Page, S. E. (1997, February). On Incentives and Updating in Agent Based Models. *Comput. Econ.*, *10*(1), 67–87. doi: 10.1023/A:1008625524072

Railsback, S., & Grimm, V. (2011). *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton University Press.

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

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.

Yuxuan, J. (2016). *The Agent-based Simulation Environment in Java*. Unpublished doctoral dissertation, University Of Nottingham, School Of Com-

puter Science.

Appendix C

Programming-paradigms in ABS

TODO: write short explanation of this paper. emphasize that it was not published but originally part of the update-strategies paper which was then split because of two different things mixed up.

Programming Paradigms in Agent-Based Simulation

Jonathan Thaler

February 22, 2017

Abstract

We compare the three very different programming languages Java, Haskell and Scala in their suitability to implement the strategies.

Keywords

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

1 Introduction

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.

2 Background

2.1 Related Research

[6] 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 the pure functional paradigm using Haskell in the field of ABS has been moderate so far. Though there exist a few papers which look into Haskell and ABS [2], [9], [6] 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 [8]. 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 so far, presenting an original novelty.

There already exists research using the Actor Model [1] for ABS in the context of Erlang [10], [3], [4], [7] 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 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. See table **??** for an overview which language and paradigm is suited for implementing which strategy. [1]

3.1 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.

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.

3.2 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 [5].

Ease of Use We initially thought that it would be suitable best for implementing the *parallel strategy*

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

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.

3.3 Multi-paradigm: Scala

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.

4 Conclusion and future research

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 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.

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

Table 1: Suitability of the languages for implementing each update-strategy: (+) *natural mapping of strategy to paradigm*, (∘) *paradigm can capture strategy but takes more effort*, (–) *paradigm not well suited to implement strategy*

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.

References

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

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

[3] 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.

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

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

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

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

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

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

[10] 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.

Appendix D

Recursive ABS

TODO: write short explanation of this paper. not published but interesting idea but not enough time to pursue. also why stopped working: don't have the right model in which it serves as a killer-features. still we are convinced that it proves one of the major benefits of FrABS: recursion is easy to implement because the language is built on it and due to the lack of side-effects.

TODO: include, need little bit of refinement