1st Year Report

Functional Agent-Based Modelling & Simulation

Jonathan Thaler (4276122)
jonathan.thaler@nottingham.ac.uk

supervised by
Dr. Peer-Olaf Siebers
Dr. Thorsten Altenkirch

June 19, 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, even 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 Modelling and Simulation (ABS) can be done using pure functional methods. We show how ABS can be implemented in the pure functional language Haskell and develop a basic underlying theoretical framework in category-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 is both specification- and implementation-language thus closing the gap between model-specification and implementation. Second: by extracting a category-theoretical view on a real-world example and showing that the pure functional code is actually an implementation of this category-theoretical view allows a new, formal approach to validation.

Contents

Chapter 1

Introduction

1.1 Background

Computer simulation is the means of imitating real-world processes over time through computational means [**?**]. 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 or the analytical solution is not tractable [**?**]. Naturally there are many different types of simulations and we focus in this thesis on Agent-Based Modelling and Simulation (ABS [1]) which 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. Epstein [**?**] identifies ABS to be especially applicable for analysing *"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 [**?**].

We informally assume the following about our agents [**?**]:

- They are uniquely addressable entities with some internal state.

[1] This abreviation will be used throughout the text and includes always the modelling *and* simulation aspect.

- 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 [**?**] where ABS is primarily used for the simulation of collective behaviour of agents to gain insight into the dynamics of a system [**?**].

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. This 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 [**?**], for our PhD we pick the fields of Agent-Based Social Sciences (ABSS) and Agent-Based Computational Economics (ACE). The former one is traditionally a highly interdisciplinary field [**?**], drawing on lots of different other areas like economics, epidemology, genetics, neurocognition, biology. It is still a young field, having emerged in the 1990's but is constantly growing and gaining significance [**?**]. 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 [**?**], [**?**]. ACE follows basically the same approach but in the field of economics to study economic processes as dynamic systems of interacting agents [**?**]. 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 [**?**]. 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 [**?**]. 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 [**?**], [**?**]. 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 [**?**].

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 [**?**] 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 programming the agents and the latter for low-level graphics [**?**]. Axelrod [**?**] 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 [**?**], [**?**]. 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 [**?**] put forward the use of *patterns* to organize objects and their interaction. Other concepts, trying to address the problems, were the SOLID principles and Dependency Injection. Although a huge step in the right direction, these concepts come with a very heavy overhead, are often difficult to understand and to apply and don't solve the fundamental problem [**?**]. To put it short: even for experienced programmers, proper object-oriented programming *is hard*. 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 object-orientation, invented by Alan Kay [2] was very different than today and 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 most notable difference between an agent and an object is that the latter one do not encapsulate behaviour activation [**?**] - it is passive. Also it is remarkable that [**?**] a paper from 1999 claims that object-orientation is not well suited for modelling complex systems because objects behaviour is too fine granular and method invocation a too primitive mechanism.

In [**?**] 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 [**?**] 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 [**?**]. The same problem was reported in [**?**] which tried to reproduce the work of Gintis [**?**]. 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 [**?**] 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. [**?**] 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 [**?**] 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 [**?**] who got access to Gintis code of [**?**]. They found that the code didn't follow good object-oriented design principles (all was public, code duplication) and - in accordance with [**?**] - discovered a number of bugs serious enough to invalidate the results. This reporting seems to confirm the

[2] `http://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented`

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 [**?**]. 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. This is a well established process in the software-industry but has not got as much attention and is not nearly as well established and easy in the field of ABS as will become evident in the literature-review. As ABS is almost always used for scientific research, producing often break-through scientific results as pointed out in [**?**], 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 and 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 we drift off into guesswork.

In the concluding remarks of [**?**] 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 motivation 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?

2. **Verification & Validation of ABS** - Is there a way for formal specification and verification which is still readable and does not fall back to pure mathematics? What exactly is the meaning of validation in ABS and is there a way to do formal validation in ABS?

Chapter 2

Literature Review

In this chapter 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. 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

The Actor-Model, a model of concurrency, was initially conceived by Hewitt in 1973 [**?**] and refined later on [**?**], [**?**]. 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 [**?**]. In the seminal work of Agha [**?**] he developed a semantic mode, he termed *actors* which was then developed further [**?**] 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 do the following *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 ones are Erlang and Scala [1].

2.1.1 Erlang

The programming-model of actors [**?**] was the inspiration for the Erlang programming language [**?**] which was created in the 1980's by Joe Armstrong for Eriksson 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 [**?**] 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 [**?**] 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 [**?**] 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 [**?**] and [**?**] 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 [**?**] 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 [**?**] focuses explicitly on

[1] The paper of [**?**] gives an excellent overview over the strengths and weaknesses of agent-based software-engineering, which can be directly applied to both of these languages.

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.

2.2 Process Calculi

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) [**?**] and Calculus of Communicating Processes (CCS) [**?**]. 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 [**?**], [**?**], [**?**], [**?**] 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 of 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 [**?**], [**?**] and in [**?**] 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*.

2.3 Discrete Event System Specification (DEVS)

Discrete Event Simulation (DES) is regarded as one of the three disciplines of simulation [2]. In it the system evolves in discrete time-steps caused by events which happen at discrete time. In between these discrete steps the system does not change - the simulation jumps from event to event. The seminal work of Zeigler [**?**] developed a formalism for specifying and analyzing such discrete event systems which allows a formal modelling and verification of such systems. DEVS has been applied to a number of problems, most notably in chemistry and biology [**?**], [**?**]. DEVS has also been connected to the π-calculus [**?**]. DEVS is of importance here because some agent-behaviour and agent-interactions can

[2] Agent-Based Modelling and Simulation (ABS) and System Dynamics (SD) being the other two.

be modelled through it, giving us a powerful formal tool for model-specification and verification.

2.4 Agent Formalisms

When looking for ways to verification and validation of ABS, it is of interest to have a more formal representation of agents and ABS. This helps to clarify the key concepts in an unambiguous, formal way which can then be applied to formal methods e.g. algebraic reasoning for proofing properties of an agent system. Wooldridge [**?**], [**?**] gives an abstract architecture for intelligent agents, introducing the concept of actions of agents influencing the environment. The book of [**?**] offers several chapters on the theoretical foundations and underpinnings of multi-agent systems [3] A very useful formalization is given in [**?**] which applies their formalism as an example to the Sugarscape model [**?**] [4]. In [**?**] the authors use the process calculi CCS and CSP for designing an agent-specification language LOTOS for formal description and verification. This is a hint that process calculi, as presented in the section above, are indeed a feasible tool for formally specifying low-level properties of agents and their interactions. The authors of [**?**] compare three formal methods for modelling and validating agent systems: Erdös a knowledge-based environment for agent programming, $Nepi^2$ a programming language for agent system based on the π-calculus implemented in LISP and I/O automata a mathematical framework used for modelling and reasoning about systems with interacting components. A more software-engineering approach is taken by [**?**] and provide an agent development line from a formal agent framework, to agent system specification, agent development to agent deployment.

2.5 Pure Functional Programming

In his 1977 ACM Turing Award Lecture, John Backus [5] fundamentally critizied imperative programming for its deep flaws and proposed a functional style of programming to overcome the limitations of imperative programming [**?**]. 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 [**?**]. The main idea behind it is that

[3] Although all of these references are primarily rooted in the multi-agent system (MAS) field - which is quite distinct from ABS as stressed in the introduction - they can as well be applied to ABS and offer some inspiration from which to draw upon.

[4] A happy coincidence because we are using Sugarscape as a major use-case for developing our methodology.

[5] One of the giants of Computer Science, a main contributor to Fortran - an imperative programming language.

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 [**?**], as one does not do imperative programming in a Turing Machine. In our thesis we selected Haskell as our functional programming language. [6]. The paper of [**?**] 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 [**?**]. The main points why we decided to go for Haskell are

- Pure, Lazy Evaluation, Higher-Order Functions 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 [7] [**?**] and is applicable to a number of real-world problems [**?**].
- 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 raised in these questions. We will now investigate whether this is really the case by looking into relevant literature.

The main conclusion of the classical paper [**?**] 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

[6] Although we did a bit of research using Scala (a mixed paradigm functional language) in ABS (see Appendix C), we deliberately ignored other functional languages as it is completely out-of-scope of this thesis to do an in-depth comparison of functional languages for their suitability to implement ABS.

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

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 [**?**] 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 [**?**], Wadler realized that monads can be used to structure functional programs [**?**]. 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 [8]. In [**?**] 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 [**?**], [**?**], [**?**] 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. It may seem that one runs into efficiency-problems in a pure functional programming language when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of [**?**] showed that when approaching this problem from a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

The concept of monads was further generalized by Hughes in the concept of arrows [**?**]. 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 [**?**] 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 programming.

One of the most compelling example to utilize pure functional programming is the reporting of [**?**] where in a prototyping contest of DARPA the Haskell prototype was by far the shortest with 85 lines of code (LoC) as compared to

[8] This is seen as one of the main impacts of Haskell had on the mainstream programming [**?**]

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 object-orientation and is definitely one of its major strength. The classic paper [**?**] gives a wonderful way of constructing an EDSL to denotationally construct a picture reminiscent of the works of Escher. A major strength of developing an EDSL is that one can reason about and do formal verification. A nice introduction how to do reasoning in Haskell is given in [**?**]. The testing-library QuickCheck [**?**], [**?**] defines 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.

2.5.1 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 [**?**], [**?**], [**?**]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [**?**]. It comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. [**?**] which discuss using functional programming for DES mention the paradigm of functional reactive programming (FRP) to be very suitable to DES. [**?**] and [**?**] 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 [**?**]. 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 [9]. In games these objects are called *game-objects* and

[9] Gregory [**?**] defines computer-games as *soft real-time interactive agent-based computer*

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 [**?**] 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 with strong emphasis on parallelism: HLogo which is a clone of the NetLogo agent-modelling framework and HDES, a framework for discrete event simulation.

Although probably the most important selling point of a pure functional language is its ease of parallelizing code due to lack of side-effects [**?**], [**?**], [**?**], [**?**] 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 [10].

2.5.2 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. The very first functional reactive language was Fran, a domain specific language for graphics and animation. At Yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa, the most recent FRP library [**?**]. The essence of FRP with Yampa is that one describes the system in terms of signal functions in a declarative manner using the EDSL of Yampa. During execution the top level signal functions will then be evaluated and return new signal functions which act as continuations. 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 [**?**].

---

simulations

[10] We did some research of implementing concurrent agents with STM as proposed by [**?**] and [**?**] in our research on programming paradigms as can be seen in Appendix E. 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.

Yampa has been used in multiple agent-based applications: [?] uses Yampa for implementing a robot-simulation, [?] implement the classical Space Invaders game using Yampa, [?] implements a Pong-clone, the thesis of [?] shows how Yampa can be used for implementing a Game-Engine, [?] 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 [?].

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.

2.5.3 Dependent Types

As already pointed out by Sweeney in [?], 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 [?], [?] and [?] 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.

2.5.4 A 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 [?] which implements a MAS in LISP.

2.6 Agent-Based Social Simulation (ABSS)

The field of social simulation can be traced back to self-replicating von Neumann machines, cellular automata and Conway's Game of Life. The famous Schelling segregation model [?] is regarded as a pioneering example. The most prominent

topics which are explored in social simulation are social norms, institutions, reputation, elections and economics.

Axelrod [**?**], [**?**] has called social simulation the third way of doing science, which he termed the *generative* approach which is in opposition to the classical inductive (finding patterns in empirical data) and deductive (proving theorems). Thus the generative approach can be seen as a form of empirical research and is a natural environment for studying social and interdisciplinary phenomena as discussed more in-depth in the work of Epstein [**?**], [**?**]. He gives a fundamental introduction to agent-based social social simulation and makes the strong claim that *"If you didn't grow it, you didn't explain its emergence"* [11] [12]. Epstein puts much emphasis on the claim that ABSS is indeed a scientific instrument as hypotheses which are investigated are empirical falsifiable: the simulation exhibits the emergent pattern in which case the model is *one* way of explaining it or it simply does not show the emergent pattern, in which case the hypothesis, that the model (the micro-interactions amongst the agents) generates the emergent pattern is falsified [13] - we haven't found an explanation *yet*. So in summary, growing a phenomena is a necessary, but not sufficient condition for explanation [**?**].

The first large scale ABSS model which rose to some prominence was the *Sugarscape* model developed by Epstein and Axtell in 1996 [**?**]. Their aim was to *grow* an artificial society by simulation and connect observations in their simulation to phenomenon of real-world societies. The main features of this model are:

- Searching, harvesting and consuming of resources.
- Wealth and age distributions.
- Seasons in the environment and migration of agents.
- Pollution of the environment.
- Population dynamics under sexual reproduction.
- Cultural processes and transmission.
- Combat and assimilation.
- Bilateral decentralized trading (bartering) between agents with endogenous demand and supply.

[11] Emergence is treated more in-depth in the Verification & Validation section.

[12] Note the fundamental constructivist approach to social science, which implies that the emergent properties are actually computable. This applies to ACE as well, which can be seen to be its most fundamental difference to general equilibrium theory of neo-classical economics which is non-constructive. When making connections from the simulation to reality (as in validation, see below), constructible emergence raises the question whether our existence is computable or not. When pushing this further, we can conjecture that the future of simulation will be simulated copies of our own existence which potentially allows to simulate *everything*. An interesting treatment of this can be found in [**?**] and [**?**].

[13] This is fundamentally following Poppers theory of science [**?**].

- Emergent Credit-Networks.
- Disease Processes, Transmission and immunology.

Because of its essential importance to this field, its complexity, number of features and allowing us to bridge the gap to ACE, we select it as the first of two central models, which will serve as use-case to develop our methods. The idea is to formally specify and then verify the process of bilateral decentralized trading because it is the most complex of the features and connects directly to ACE.

In 2013 Epstein introduced the *Agent_Zero* model [**?**] in which the author approaches the generative social sciences from a neurocognitive perspective [14]. *Agent_Zero* is an agent which is endowed with emotional/affective (emotional/gefühlsbezogen), cognitive/deliberative (wahrnehmung/abwägend) and social modules which are all interconnected and interact with each other. Also Agent_Zero is always part of a social network through which it is influenced by other Agent_Zero and can influence them. The core behaviour Epstein wants to "grow" in this model is *"the person who feels no aversion to black people, who has never had any direct evidence or experience of black wrongdoing [...], and who yet initiates the lynching"* [15]. The central concept of the model is the one of *dispositional contagion* which allows to replicate and simulate the following scenarios (amongst others):

- Fight vs. Flight
- Replicating the Latané-Darley experiment
- Growing the 2011 Arab Spring
- Jury processes
- Prices and seasonal economic cycles
- Mutual escalation spirals

We select *Agent_Zero* as our second central model serving as use-case to develop our methods because of its in ABSS and offers a very interesting use-case to apply various networks as presented in the ACE section [16]. As Epstein only looks at a network of three agents, the idea is to investigate the effect of various types of networks as presented in the literature-review section on ACE with much more than three agents on the model.

[14] Epstein termed this work Volume III in the triology on generative social science. Volume I is the Sugarscape book mentioned above [**?**]. Volume II is a collection of papers published in the book [**?**] which applied agent-based modelling to the fields of economics, archaeology, conflict, epidemiology, spatial games and the dynamics of norms.

[15] [**?**], page 2

[16] In a recent work [**?**] Epstein offers a range of new research directions for Agent_Zero, most notably new interactions, empirical testing, replication of historical episodes and formal axioms for modular agents. We include it for completeness but it does not offer fundamentally new insights to Agent_Zero neither does it approach the lack of a deeper treatment of the influence of networks in the model.

2.7 Agent-Based Computational Economics (ACE)

The field of economics is an immensely vast and complex one with many facets to it, ranging from firms, to financial markets to whole economies of a country [?]. Today its very foundations rest on rational expectations, optimization and the efficient market hypothesis. The idea is that the macroeconomics are explained by the micro foundations [?] defined through behaviour of individual agents. These agents are characterized by rational expectations, optimizing behaviour, having perfect information, equilibrium [?]. This approach to economics has come under heavy critizism in the last years for being not realistic, making impossible assumptions like perfect information, not being able to provide a process under which equilibrium is reached [?] and failing to predict crashes like the sub-prime mortgage crisis despite all the promises - the science of economics is perceived to be detached from reality [?]. ACE is a promise to repair the empirical deficit which (neo-classic) economics seem to exhibit by allowing to make more realistic, empirical assumptions about the agents which form the micro foundations. The ACE agents are characterized by bounded rationality, local information, restricted interactions over networks and out-of-equilibrium behaviour [?]. Tesfatsion [?] defines ACE as *[...] computational modelling of economic processes (including whole economies) as open-ended dynamic systems of interacting agents.*. She gives a broad overview [?] 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 behaviours 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?

Other works which investigate ACE as a discipline and discuss its methodology are [?], [?], [?], [?].

During the reading-process we became particularly interested in the dynamics of bilateral decentralized bartering [17]. The reason for this is that the Sugarscape model [?], mentioned in the previous section on Social Simulation, implements this bilateral decentralized bartering and looks at the out-of-equilibrium

[17] The Sugarscape book [?], 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

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 [**?**], [**?**], [**?**] and [**?**].

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 [18]. Introductory texts to market microstructure are [**?**], [**?**] and [**?**]. A highly interesting research using ABS for simulating the NASDAQ market was done in [**?**]. The authors where approached by NASDAQ 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 on using ABS in finance and stock markets are [**?**], [**?**], [**?**] and [**?**]. Another sub-field is autonomous and automated trading agents [**?**], [**?**].

Another topic highly important to ACE is the one of networks [**?**]. Although it is not unique to ACE or 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 [**?**]. Networks also play a very important role in Social Simulation and both Sugarscape [**?**] and Agent_Zero [**?**] incorporate them in their model, so this is another bridge from ACE to Social Simulation. In Sugarscape they occur as emerging neighbour-, genealogical-, cultural-, credit- and disease transmission-networks and in Agent_Zero networks define the influence of agents amongst each other. A short-coming of Agent_Zero is that only a network of three agents is considered - it would be interesting how dynamics unfold in other types of networks. The books of [**?**] and [**?**] give a very broad and in-depth overview over social networks with focus on economics. The paper of [**?**] is a thorough review of the field of networks focusing on small-world effects, degree distribution, clustering, random graph models, referential attachment and dynamic processes on networks. A category-theoretical approach to networks is given by Spivak in [**?**], which may bridge the gap to functional programming and its category-theoretical approach.

To conclude, 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 - together with Social Simulation - as a use-case to develop the tools and apply them directly to ACE to demonstrate the usefulness and benefit of the new tool.

[18] 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 [**?**]. In the preparation phase we did quite some reading in this field inspired by [**?**] and [**?**] 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: [**?**], [**?**], [**?**]. A broad overview of Market Design using Agent-Based Models is given in [**?**].

2.8 Verification & Validation of ABS

Verification & Validation are, generally speaking, independent processes to check whether a product meets its requirements and fulfills its intended purpose. Here we focus explicitly on software verification & validation where we identify *Verification* to be the process of checking whether an implementation matches a given specification without any bugs or missing parts and *Validation* to be the process of checking if the implementation meets high level requirements. To put short according to Boehm [**?**] the difference between verification & validation is that verification tries to answer "are we building the product right?" and validation "are we building the right product?". For (most of) the software built in the industry in well-defined software-development processes with its own quality control and quality assurance to answer these questions is rather straight-forward. Numerous techniques like checklists, software-tests, integration-tests,... have been developed to deal with Verification & Validation. The question is how the above applies to ABS and from reviewing the literature [19], it becomes evident that in the context of ABS it is a rather difficult process as there exist no straight-forward way [**?**] and is still open research [**?**] [20].

[**?**] clarifies on this subject a bit and defines verification to be "the process of determining that the equations are solved correctly [...] does the model do what we think it is supposed to do?" and validation to be "the process of determining that we are using the correct equations" - basically a reformulation of the questions posed by Boehm [**?**] above. They add on these definitions by adding the concept of a model: verification is then "the process of determining that a computational software implementation correctly represents a model of a process" and validation is "the process of assessing the degree to which a computer model is an accurate representation of the real world from the perspective of the models intended applications". A fundamentally observation [**?**] make is that ABS may blur the boundary between verification and validation. This is due to the problem of plausibly deciding how the micro-behaviour influences the macro-behaviour when changing e.g. a variable which drives the micro-behaviour. For this to decide we have to turn to validation, where we check if the model is a plausible image of the real-world process. [**?**] makes the critical point that the dynamics of a model in ABS are almost always so complex, that the creator of the model is not able to exactly explain what the deeper reasons are for them and which aspects of the model are responsible for their exhibition. If this were so, one would not need to resort to simulation. This implies that one can not know in advance what exactly to expect and which part of the emergent behaviour of the system connects to which local interaction amongst agents. Also it is very hard to check if the emergent behaviour is not due to some bug in the implementation.

[19] Leigh Tesfatsion has, as part of her internet-presence on ACE, set up a whole site devoted just to this topic where she cites major references `http://www2.econ.iastate.edu/tesfatsi/empvalid.htm`. Most of the literature we have investigated is drawn from there.

[20] This work gives an in-depth discussion and treatment of verification, validation and testing of complex and large-scale simulations from an engineering point of view.

Also an important question raised in [**?**] is the one how the time-stepping of ABS relates to time in reality [21]. Concluding, the authors follow the Popperian position 'that validation can never be proof'. They stress that an important part of validation is a clear description of what is being explained (My question: can this be only natural language e.g. could we formalize this? if yes, we can formalize validation!).

The issue of validation & verification is also very closely related to the problem of replication. In the paper [**?**] Wilensky discusses the issues with replicating a model and gives advices to modellers what details (e.g. order of events) to publish to support replication of models. They recommend to make pseudo-code publicly available and to converge to a common standard form for model publication in the long run. This is our approach: an EDSL can act both as a pseudo-code specification which is in fact already runnable code and due to its declarative and concise nature can act as a full description.

The paper of [**?**] explores problems with empirical validation of agent-based models. They identify six issues and presents a novel taxonomy to classify the approaches developed by the ABS community so far in tackling these issues. The authors then discussed three approaches to ABS validation and showed how to apply them.

In this thesis we aim for replicating both the Sugarscape [**?**] and Agent_Zero [**?**] models. In doing so we also perform verification & validation on these models, or at least on parts of them. Also we aim to develop new methods of verification and validation of ABS based on theses models as use-cases.

2.8.1 Verification

Axelrods seminal paper [**?**] establishes three ways of comparing models: same numerical outputs, same distributions of numerical outputs, same correlation of outputs: if x is increased in both models and y increases in the original then it should also be increased in the replicated model. [**?**] claim that the last property is rather too weak and extend it to be 'if input variable x is increased in both models by a given amount, the distribution observed in the changes in output variable y should be statistically indistinguishable'.

The cooperative work of [**?**] gives insights into verification of computational models, in a process what they call "alignment", determining whether two models deal with the same phenomena or not. The authors tried to see whether the more complex Sugarscape model can be used to reproduce the results of [**?**]. 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 [**?**] which only focuses on transmission of these cultural tags. Although interesting the question if two

[21] My paper on Update-Strategies clarifies on this a bit: although we perceive time in reality as a continuous flow, the way we *implement* time in a simulation is always in steps, be it discrete- or continuous-time (see Appendix D).

models are qualitatively equivalent is not what we want to pursue in our thesis as it requires a complete different direction of research.

2.8.2 Validation

Wilensky [**?**] defines model validation to be the process of determining whether the simulation explains and corresponds to phenomenon of the real world. [**?**] define validation of a model to check if it is consistent with the intended application of the model, how well the model captures its empirical referent.

Carley [**?**] is a comprehensive reference focused explicitly on validation and provides information and techniques for it. The author distinguishes between six types of validation:

1. Conceptual - is the underlying theoretical model adequate to characterize the real world?
2. Internal - is the computer code correct, free of coding errors [22]?
3. External - is the model adequate and accurate in matching the real world data?
4. Cross-Model - is the output of the model qualitatively the same as another model [23]?
5. Data - is the data adequate and accurate for addressing the problem?
6. Security - has the model been tampered with [24]?

It becomes clear that validation is concerned with establishing the relationship between the simulated model and its real-world counterpart - the system / thing / object being simulated. The question is what exactly is compared and whether there is a view which connects both of them. This brings us to the central concept of emergence.

2.8.3 Emergence

An essential aspect of ABS is the one of emergent properties of a simulation: the global dynamics the system exhibits under simulation emerges from the local interaction of its parts where these local interactions were not designed to force the system towards this emergent property - the whole system exhibits properties the parts don't have, a collection of interacting systems shows collective behaviour [**?**] or simply *the whole is more than the sum of its parts.* When doing verification - checking the relationship between the simulated model and its

[22] This may be confused with the established definition of verification. The question is to which kind of correctness this definition refers: a program can be correct in the way that it has no bugs e.g. causing no crashes, or it is correct in the way that it implemented the model as it was intended.

[23] An imporant example for this is the work of [**?**], see above

[24] We don't understand what is really meant here.

real-world counterpart - we need to check if the emergent properties of both systems match or are qualitatively the same. For this we need *some* formalization of emergence.

A non-formal view on emergence from the social sciences is given by Gilbert [**?**]. He defines emergent behaviour to be that which cannot be predicted from knowledge of the properties of the agents except as a result of simulation and conjectures that this definition gives an explanation for the popularity of simulation for researching complex adaptive systems. Gilbert then raises the question if emergence can be formalized.

Baas [**?**], [**?**] attempts a highly intriguing formalization of emergence. In [**?**] he approaches emergence from a category theory perspective [25] where systems are represented as objects and their interactions as morphisms. Baas introduces the concept of an observer [26] and observational mechanisms which can be thought of a functor, mapping from one category to another - emergence is a holistic structure which can not be decomposed into its parts. He then distinguishes between two kinds of emergence: deducible or computational emergence for which an algorithmic explanation exists and observational emergence for which none such explanation exists. It seems that the work of Baas on emergence could be used to drive our ambitions in using category-theory as underlying foundation for ABS and model-specification and connect it to the foundations of and transfer it to pure functional programming. This could close the second gap which exists between the emergent-property of the model specification and the real-world system through means of functional programming.

2.9 Category-Theory

The field of Category-Theory formalizes mathematical structure by abstracting away from internal structure of mathematical ojbects and only looking at relations between them. It developed out of algebraic topology in the 1940s by the work of Samuel Eilenberg and Saunders Mac Lane with the goal of understanding the processes that preserve mathematical structure. The central concepts are collection of objects and arrows, called morphisms which are structure preserving mapings between these objects. An example for a category is the category of sets with sets being the objects and the arrows are functions from one set to another. It is important to make it clear that in this case an objects is a set which itself consists of elements but in category-theory one does and must not look at these elements - only the extensional properties of the object are relevenat: how does it relate to other objects through morphisms.

[25] It is no coincidence that the formalization of emergency is given by Baas, a mathematician in the field of Algebraic Topology - a field out of which category theory was born. This may be a hint that a category theory approach to ABS may indeed work and that it allows to abstract from the simulated model and the real-world system and show that the simulated model is indeed an instance of the real-world system.

[26] Note the importance of an observer for emergence: an essential question is when to believe a model and how to interpret its results and its emergent behaviour - something highly dependent on an observer

The classic introductory text for computer scientists is [**?**]. A more comprehensive text with more emphasis on applications to the (computer) scientist is [**?**].

In the work of [**?**] the author has shown that category-theory can be used to link concepts of computation, physics and logic. As our thesis is multidisciplinary our aim is also to use category-theory to abstract from these concrete fields and derive common structure.

Computation With the advent of monads in functional programming, the interest in category-theory surged and it was discovered that many computational concepts can be expressed through category-theory.

ABS So far only two papers looked into category-theoretical approaches to agent-based models and simulating [**?**], [**?**] but none of them is really satisfying as they lack . The lack of a proper treatment of ABS and 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.

Emergence We already mentioned the work of Baas [**?**], [**?**] on formalizing emergence through category-theory in the section on Verification & Validation.

Concluding, we can say that the essence of category-theory is to derive structural concepts and compare them. This gives us a high-level structural view to compare concepts which may seem be completely different at first and see it all through one lens. Thus our central hypothesis is that if we formulate the real-world phenomenon, the model-specification, ABS and the functional implementation in category-theory we may be able to show that they all represent the same thing. So we arrive at the very essence of validation by showing that the functional program / simulation is indeed a faithful implementation of the real-world phenomenon. To our knowledge such an approach to validating ABS has not been attempted yet.

Chapter 3

Reflecting the Literature

In this chapter we reflect the literature we have investigated in the previous chapter to identify a research-gap and derive a vision for our research which will be clarified in the next chapter on aims and objectives. When looking at the literature it seems that the relevant concepts for approaching the problems mentioned in the introduction are there, but that they are distributed over the various topics.

3.1 Actor Model

The core concept of the Actor Model is that actors are processes which have *share nothing* semantics: there exist no aliases, through which actors can implicitly change the state of another actor through side-effects. The only means of interaction is through message passing in which case data is copied and nothing is shared. This approach guarantees an explicit data-flow with localized state and is the original vision Alan Kay had when he conceived object-orientation [1] and exists in its most faithful implementation in Erlang [2]. Due to its share nothing semantics, actors are assumed to run in parallel, synchronizing only on messaging which would allow for a very high number of actors.

The core weakness is that message passing is asynchronous and inherently unreliable and that actors are only reactive. This is a very problem in ABS as it leads to non-reproducible simulations as the randomness does not rely on model-inherent properties like random-number generators but on exogenous parameters. As pointed out by [**?**] on agent-based software-engineering, the problems are that patterns of the interactions are inherently unpredictable and that predicting the global system behaviour is extremely difficult. This observation is in unison with the results of my paper on update-strategies (see Appendix D) where we showed that a truly agent-based solution (actor-strategy) leads to

[1] See `http://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented`

[2] Although Scala also comes with an Actor-Library, it allows sharing by sending mutable messages and references thus violating the locality of state.

non-deterministic results due to inherent concurrency. A truly concurrent implementation is only useful when the model-semantics are concurrent as well, where the ordering of events does not matter (as shown in the Heroes & Cowards Game). This is very rarely the case thus an implementation of parallel or sequential semantics are favourable (and are sufficient when event-ordering does not matter). Also the asynchronous nature of messaging makes it difficult to handle cases where synchronous messaging is required e.g. in trading between two agents. This would require to synchronize agents through a central auctioneer, thus rendering the parallelism advantage void. Besides, we don't have any model of a global time, which is almost always necessary in ABS and could only be introduced by a global time-agent, synchronizing all agents.

3.2 Process Calculi & DEVS

Process calculi are nice for specifying and verifying concurrent computations but are too cumbersome to fully implement a 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 [3]. On the other hand, process calculi are used in the industry for verification purpopses so 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 π-calculus and then apply algebraic reasoning. As emphasised in the literature-review, no 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 [**?**], which strengthen our argument. DEVS is, although of very different nature than process calculi, on the same low-level thus the same what we said about process calculi apply to DEVS as well.

3.3 Agent Formalisms

There exist a lot of agent formalisms in the field of multi-agent systems (MAS) which allow the formal and high-level specification of such a system. Some make use of low-level features like process calculi and algebraic reasoning. Our intention in this PhD is not to deliver a new agent formalism but to draw inspiration from them and bringing their strengths - a high-level specification of interacting agents - to a real programming language.

3.4 Pure Functional Programming

As pointed out in the literature-review at length, the strengths of pure functional programming (as in Haskell) are manifold and overcome the problems of object-

[3] It was shown by Milner [**?**] 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.

oriented programming we indicated in the introduction [4]. The most powerful aspect is obviously its *purity*: it is explicit about effects (through monads) and has thus an explicit data-flow. Its composability achieved through higher-order functions and lazyness is superior to objects because of the loose coupling between data and code. The declarative style allows to easily implement EDSLs for a given problem which makes reasoning possible, also due to the lack of implicit side-effects. Finally the static type system is a powerful tool to create contracts and specifications in code which supports reasoning and increases correctness and can be seen as a form of additional documentation [5].

Of course pure functional programming has also its weaknesses. The main issue in a lazy functional programming language is the difficulty of predicting space behaviour, resulting in *space-leaks*, which is very hard even for experienced programmers [**?**]. 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 [**?**]. 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. 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 [**?**], [**?**] - the works on game-programming in Yampa mention a similar performance-problem. Another weakness is composition of Monads: although monads are seen a benefit to pure functional programming by allowing to making side-effects explicit, so far multiple monads do not compose in a nice, modular way and this issue is still open research [**?**].

3.5 Combining

Having reflected the approaches we can now derive what is needed for handling the problems mentioned in the introduction: we are combining the strengths of

[4] This does not mean that it makes object-oriented programming obsolete or that it can be applied equally well to all domains which have been assumed to be classic object-oriented domains - here we are just interested in ABS and we claim the pure functional way is very well suited for it - see below. Although attempts exist in bringing pure functional programming to e.g. GUI- and Game-programming, we think that especially these two fields are better suited to object-orientation because in both fields side-effects are fundamental and pervasive in every aspect of the domain and thus utterly difficult to isolate in a pure-functional way (which also allows for high-performance, see below).

[5] Of course to some, a static type system is just an annoying obstacle in writing code, but when going in the direction of verification and validation, it is a mandatory tool and can not be left aside.

the approaches. From process-calculus, DEVS and agent formalisms we take the concept of a domain-specific language which allows to reason about the system and apply algebraic laws in manipulating it. Also we hope to make use of the π-calculus and DEVS for verification purposes although in which way precisely they can be made of use to ABS is unclear and will be part of the research. From the actor model we take the concept of processes which interact with each other by shared-nothing messaging. When applying these to pure functional programming in Haskell we arrive at an ideal combination. The purity allows us to implement deterministic, synchronized and reliable messaging with shared-nothing semantics. The declarative nature and its static type-system allows us to implement an EDSL inspired by the actor model and the formalisms mentioned above. Building on top of a FRP library like Yampa gives us the necessary time-semantics for ABS and allows us to formulate agents as processes, implemented as signal-functions (see chapter on functional reactive ABS).

3.6 Identifying the Gap

Functional programming in this area exists but only scratches the surface and focus only on implementing agent-behaviour frameworks like BDI and focus only on MAS. 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. Also there basically exists no approach to ABS in terms of category-theory which form the basis of pure functional programming.

Chapter 4

Aims and Objectives

4.1 Aims

As has become evident from the reflections on the literature 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 for undertaking this research by itself as it - quite surprisingly - hasn't been done yet and deserves a thorough treatment on its own. Maybe this can be seen as a hint that ABS lacks a level of formalism which we hope to repair with our thesis. Also 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 shareable with the community. 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 [**?**] we opt for a similar direction. Having Haskell as the implementation language instead of an object-oriented one like Java allows us to build an EDSL for ABS which can act both as specification- and implementation-language, closing the gap between specification and implementation. This would give us a way of formally specifying the model but still in a more readable and tractable way than pure mathematics. This form of formal specification can then act easily as a medium for communication between team-members and to the scientific audience in papers as it is both specification- and implementation-language. This point seems to be quite ambitious, but seeing it as a hypothesis we will see how far we can get with it. Also central to our verification approach will be the QuickCheck library. We will use it to formulate model-specifications and specifications of the FrABS itself directly in code. It would be of interest to put pure functional ABS on a firm theoretical ground by developing a category-

theoretical view on ABS. This could potentially, give a deeper insight into the structure of agents, agent-models and agent-based simulation and serves as the basis for the pure functional implementation and as a high-level specification tool for agent-models. In the literature-review we have seen that there exist category-theoretical views on models for verification, a direction definitely worth going to. Because of the category-theoretic foundations of Haskell it may be the case that we also close the gap between conceptual model and implementation thus making a huge advancement in validation. With the powerful verification tools developed we can approach verification from a new perspective. When implementing a simulation one follows roughly three steps: 1st observing the real-world example, 2nd create a formal model specification and 3rd implement the simulation. In verification we need to show that the implementation captures the essence of the aspect of the real-world sample we want to simulate. Our novel idea is that first we show that the implementation of the simulation is a correct implementation, following the formal model specification. This becomes trivial when the formal model-specification is already the implementation-language. Second we derive a category-theoretical representation both of the real-world example *and* the model and then show that they are equal. The implication is that then the simulation implementation must be a faithful representation of the real-world example. Our aim is to primarily focus on the decentralized bilateral trading & bartering process in Sugarscape, building on the functional model of exchange of [**?**] and research our new verification- and validation-methods based on this problem. This also bridges the gap to ACE and economics as it is a well researched topic with lots of formal theory to it. We may come to the conclusion that the Sugarscape approach to decentralized bilateral trading & bartering may be too complicated and that we have to resort to a simpler model which is equal in explanatory power. This danger is indeed a real one because of the endogenous demand & supply which is driven by sugar- and spice-harvesting which in turn depends on many additional properties and behaviour like vision, metabolism, environment,...

4.2 Hypotheses

Based upon our aim we derived the following hypotheses which will guide us as bold, motivating claims to drive forward our research.

4.2.1 Feasibility of functional ABS

Functional reactive programming (FRP) in the implementation of Yampa is a useful tool to implement pure functional ABS in Haskell.

Building on FRP it is possible to implement the Sugarscape and the Agent_Zero models with less lines of code and more expressiveness

than corresponding Java or NetLogo code [1]

4.2.2 Verification

We can develop an EDSL on top of the functional ABS library which is both specification- and implementation-language and thus closes the gap between specification and implementation. This EDSL can then be used to concisely specify and communicate a model with high expressiveness.

QuickCheck allows us to formulate specifications of a model directly in code, built on the EDSL and check them.

4.2.3 Validation

If we find a category-theoretical description of the real-world concept and develop our functional ABS model after it or show that it follows this description as well, then it must follow that our implementation *is* indeed equivalent to the real-world concept and thus implicitly valid.

QuickCheck can be used for validation as well.

4.3 Objectives

Based upon the aims and hypotheses we define the following four objectives we want to achieve in our research. The objectives are ordered sequentially in the way they will be undertaken in the course of the remaining PhD.

4.3.1 Functional reactive ABS (FrABS)

The first goal is to implement a library for pure functional ABS in Haskell, building on the FRP paradigm using Yampa. We term the combination functional reactive agent-based modelling & simulation: FrABS. The driving use-cases for building this library will be both the Sugarscape- and Agent_Zero model. The resulting library implements a very rudimentary EDSL for FrABS and will show that functional ABS is indeed very possible, elegant and more concise than object-oriented solutions.

[1] There exists a Java-implementation of SugarScape `http://sugarscape.sourceforge.net/` which will be subject of investigation during this PhD. NetLogo comes with three Sugarscape models which implement but only the first very basic features. We couldn't fine a *full* NetLogo implementation freely available.

4.3.2 Functional Verification

The next step is to take the previously developed FrABS library and refine its EDSL to a point where it can be used as a specification language. We test this by giving specifications of all the full Sugarscape rules as described in the book [**?**]. In the next step we will then turn towards an in-depth investigation of the decentralized bilateral trading & bartering process. We investigate the potential of using QuickCheck for formulating and testing specifications and how far we can get with reasoning about dynamics and equilibria using our EDSL, QuickCheck and Haskell.

4.3.3 Category Theory view on ABS

After having established the verification we try to derive an ABS representation in category-theory. By mapping the concepts of ABS to category-theory we hope to gain a deeper understanding of the deeper structure behind agents, agent-based models and agent-based simulations.

4.3.4 Functional Validation

The final step is then an attempt to combine the previous steps to achieve formal validation of the decentralized bilateral bartering process. We will try to derive a category-theoretical model of real-world bartering and compare this to a category-theoretic view on our implementation. If they match, we have showed that the implemented model-specification - which is at the same time Haskell code - is a valid and faithful representation of the real-world process.

4.4 Research Questions

We present the research questions at the end of this chapter, because they contain the essence of the previously explained arguments and hypotheses. The questions are ordered according to their specific topics.

4.4.1 ABS

- Can we derive a category-theoretical view on functional ABS?
- Can we represent emergent properties of a real-world model in category-theory and encode this in Haskell, thus closing the validation-gap?

4.4.2 Functional Programming

- How can FRP (as in Yampa) be applied implementing functional ABS?
- How can QuickCheck be made of use to functional ABS verification and can it be used for validation as well?

4.4.3 ABS & Functional Programming

- How can we reason about the dynamics and equilibria of the decentralized bilateral trading & bartering?
- How does an EDSL for functional ABS, built on FRP looks like? Can we really close the gap between specification and implementation?

Chapter 5

Work To Date

Here we give a concise overview over the activities performed since the beginning of the PhD. We also attended several courses, which are listed in Appendix B

5.1 Papers Submitted

5.1.1 Update-Strategies in ABS

This paper, which is attached in Appendix D, is the essence of the prototyping and experimenting with bringing ABS to Haskell and comparing the solution to Java. It covers one of the most fundamental aspects of ABS, the issue of how agents are updated. When reading literature, it became evident that there is are inconsistent terminology for speaking about this issue, which also lacks precision and misses important details. We derive these missing details, develop a new terminology and present the four possible ways to update agents and discuss the differences they make. The main conclusion of the paper is that the way agents are updated must reflect the semantics of the model - this may be obvious in the first place but when looking closer into the intricacies much care must be taken due to the details and implications of each update-strategy. We submitted the paper to the Social Simulation Conference 2017 (SSC 2017) [1], which will take place from 25-29th September 2017 in Dublin. For gaining the insights in update-strategies we wrote quite a bit of code: for Java and Haskell [2] we implemented the Heroes & Cowards game, Spatial Prisoners Dilemma and SIRS Model in all four update-strategies. For Scala we only implemented Heroes & Cowards using the actor-strategy. [3].

[1] `http://www.sim2017.com/`

[2] We didn't use FRP at this point and implemented all without a basic library.

[3] The code of all implementations can be accessed from `https://github.com/thalerjonathan/phd/tree/master/coding/papers/iteratingABM`

5.2 Paper Drafts

5.2.1 Programming Paradigms and ABS

This paper, which is attached in Appendix E, covers the very essence of the software-prototyping conducted in the first months of the PhD. The goal was to see how well the very three different programming paradigms of object-orientation (Java), pure functional (Haskell) and mixed paradigm with actor (Scala) are suited for implementing ABS. Originally this work was part of the submitted paper on update-strategies but was extracted into a separate paper. It turned out that although there is a strong connection between these topics, they should be split into two papers for purpose of clarity, supporting the process of submitting [4] and arriving at potentially a second paper. The work lies dormant at the moment but could be picked up in the future and be developed in a full-fledged conference paper [5] as there are already quite strong basics there. What we would like to do when continuing work on this paper is a closer look in using Software-Transactional-Memory (STM) for concurrent ABS.

5.2.2 Recursive ABS

The idea for this paper arose from my idea of *anticipating agents*, which can project their actions in the future. Because this paper is not as polished as the draft for programming paradigms, we opted not to include it as an appendix and only give its basic ideas and results for the experiments conducted so far. Note that we were not able to find any research regarding recursive ABS [6]. In Recursive ABS agents are able to halt time and 'play through' an arbitrary number of actions, compare their outcome and then to resume time and continue with a specifically chosen action e.g. the best performing or the one in which they haven't died. More precisely, what we want is to give an agent the ability to run the simulation recursively a number of times where the this number is not determined initially but can depend on the outcome of the recursive simulation. So Recursive ABS gives each Agent the ability to run the simulation locally from its point of view to anticipate its actions in the future and change them in the present. We investigate the famous Schelling Segregation [**?**] and endow our agents with the ability to project their actions into the future by recursively running simulations. Based on the outcome of the recursions they are then able to determine whether their move increases their utility in the future or not. 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

[4] A paper should not confuse and mix two topics, although they might be very close. This helps both the reviewers and readers in understanding and accepting the paper.

[5] It is unclear to which conference it could potentially be submitted but we think SSC is suitable because it discusses these concepts on a very high level and refrains from code listings.

[6] We found a paper on recursive simulation in general [**?**] which focuses on military simulation implemented in C++. Its main findings are that deterministic models seem to benefit significantly from using recursions of the simulation for the decision making process and that when using stochastic models this benefit seems to be lost.

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 agent were satisfied - the move of a single agent caused the system to become completely unstable and depart from its near-equilibrium state to a highly volatile and unstable state.

This approach of course rises a few questions and issues. The main problem of our approach is that, depending on ones view-point, it is violating the principles of locality of information and limit of computing power. To recursively run the simulation the agent which initiates the recursion is feeding in all the states of the other agents and calculates the outcome of potentially multiple of its own steps, each potentially multiple recursion-layers deep and each recursion-layer multiple time-steps long. Both requires that each agent has perfect information about the complete simulation *and* can compute these 3-dimensional recursions, which scale exponentially. In the social sciences where agents are often designed to have only very local information and perform low-cost computations it is very difficult or impossible to motivate the usage of recursive simulations - it simply does not match the assumptions of the real world, the social sciences want to model. In general simulations, where it is much more commonly accepted to assume perfect information and potentially infinite amount of computing power this approach is easily motivated by a constructive argument: it is possible to build, thus we build it. Another fundamental question regards the meaning and epistemology behind an entity running simulations. Of course, this strongly depends on the context: in ACE it may be understood as a search for optimizing behaviour, in Social Simulation it may be interpreted as a kind of free will: the agent who is initiating the recursion can be seen as 'knowing' that it is running inside a simulation, thus in this context free will is seen as being able to anticipate ones actions and change them. When talking about recursion it is always the question of the depth of the recursion and because as we are running on computers we need to terminate at some point. Accelerating Turing machines (also known as Zeno Machine) are theoretically able to calculate an infinite regress but this raises again epistemological questions and can be seen as having religious character as discussed e.g. in Tiplers Omega Point, Bostroms simulation argument [**?**] and its theological implications [**?**]. So the ultimate question this research leaves is what the outcome would be when running a recursive ABS on a Zeno Machine/Accelerated Turing Machine? [7]

At the moment this idea lies dormant as the intention was just to develop it far enough to give a proof-of-concept and see some results. Having achieved this we arrived at the conclusion, that the results are not really ground-breaking. This stems from the fact that Schelling segregation is not the best model to demonstrate this technique and that we are thus lacking the right model in which recursive ABS is the real killer-feature. Also to pursue this direction further and treat it in-depth, would require much more time and give the PhD a complete different spin. Still it is useful in supporting our move towards

[7] Anyway this would mean we have infinite amount of computing power - I am sure that in this case we don't worry the slightest about recursive ABS any more.

pure functional ABS as we are convinced that recursion is comparably easy to implement because the language is built on it and due to the lack of side-effects [8].

5.3 Functional Reactive ABS

This is the first of the four objectives and is our approach of implementing pure functional ABS in Haskell using the FRP paradigm using Yampa. We are not yet in the process of writing a paper and have thus written an overview of the approach in Appendix C, which discusses the approach, implementation-details and insights so far. Our intention so far is to write a (journal) paper from it and submit it to the Trends in Functional Programming (TFP) 2018 [9] which will be held at some time in June 2018. We have already implemented basics of the library and have used it to build the Sugarscape- and Agent_Zero models and Schelling Segregation, SIRS and Recursive ABS [10].

5.4 Reports

5.4.1 Haskell Communities and Activities Report (HCAR) May 2017

We wrote a new entry for the HCAR May 2017, which tries to compile and publish novel and on-going ideas in the Haskell community. It is freely available under `https://www.haskell.org/communities/05-2017/html/report.html`. We hope that our idea and the work of our PhD gets a bit more attention and may start some discussions with people interested in this work.

5.4.2 1st Year Report

This document.

5.5 Prototyping in Haskell

To gain more insight into Haskell, how to approach ABS in a pure functional language and to learn parallelism and concurrency in Haskell the following prototypes were developed [11]:

- Heroes & Cowards

[8] Actually implementing it was *really hard* but we wouldn't dare to implement this into an object-oriented language or into an object-oriented ABS framework.

[9] `http://www.tifp.org/`

[10] The code can be accessed from `https://github.com/thalerjonathan/phd/tree/master/coding/libraries/frABS/src`

[11] The code can be accessed from `https://github.com/thalerjonathan/phd/tree/master/coding/prototyping/haskell`

- SIRS Model
- Spatial Prisoners Dilemma
- Wildfire
- 2D-Graphics rendering in OpenGL and Gloss
- Agents running in IO-Monad, STM, parallel and pure

5.6 Talks

So far only two talks were given. The first one was a presentation of the ideas underlying the update-strategies paper at the IMA - seminar day. The second was presenting my ideas about functional reactive ABS to the FP-Lab Group at the FPLunch.

Chapter 6

Future Work Plan

In this chapter we break down the the four main objectives we have identified in Chapter 4 into small working packages, present mile-stones and holidays, relevant conferences, papers we want to publish (either conference or journal). A Gantt-Chart, reflecting all the details is found in Appendix A in Figure A.

6.1 The Four Objectives

6.1.1 Functional reactive ABS (FrABS)

The main goal is to implement a library to be release on Hackage which implements functional reactive ABS (FrABS) using Yampa. As use-cases the Sugarscape- and Agent_Zero-Model should be implemented. The outcome should be published in a paper (see below) which marks the end of this objective. The time-frame will be until around February 2018.

6.1.2 Functional Verification

The main goal of this objective is to implement our novel idea of verification by refining the EDSL of FrABS to a point where it can be used both as specification- and implementation-language. As use-case we pick the decentralized bilateral bartering as specified in the Sugarscape model. Of central focus will be the research of how to use QuickCheck in the verification process for both the use-case and the FrABS library. The outcome should be published in a paper (see below), summing up the research of this objective.

6.1.3 Category Theory view on ABS

This is an intermediary objective and serves only for studying basics of category-theory and apply it to ABS to gain a deeper understanding of the deeper structure behind agents, agent-based models and agent-based simulations. The out-

come should be published in a paper (see below), summing up the research of this objective.

6.1.4 Functional Validation

In the final objective we built on all the previous research and combine it into a final paper to show the novel approach to verification and validation using FrABS, QuickCheck and category-theory. Again the use-case will be the decentralized bilateral bartering process as specified in the Sugarscape model but supplemented by theoretical work on bilateral decentralized bartering and real-world examples. Also the category-theoretical view on ABS and on decentralized bilateral trading will be incorporated into this work. The outcome will be a journal-paper to be submitted in the field of ABS (see below).

6.2 Planned Papers

We plan for three papers with at least one of them to be submitted to a journal. This follows the split of the PhD into its objectives: the first concentrates on implementation, the second on verification, the third on category-theory in ABS and the fourth on validation.

6.2.1 FrABS - Towards pure functional programming in ABS

This paper will present the pure functional approach to ABS we have taken and outlined in Appendix C. It will describe the combination of FRP & ABS, the EDSL built on top and gives some examples of specification and implementation of the SIRS and Sugarscape model. For publishing we think of two strategies: either we focus on a conference in the field of ABS or a journal for functional programming. We opt rather for a conference paper, presenting it to an ABS audience because the ABS-community is our intended target - the functional programming people don't have to be convinced that FP is great. Also it is unclear if a functional programming journal accepts the interdisciplinary work (I regard the FP guys to be still open minded but they may be rather conservative compared to ABS community?). The target-journal/conference is yet to be determined.

6.2.2 Verification in pure functional ABS

In this paper we will discuss the verification method we have developed using out FrABS implementation and QuickCheck. We describe the decentralized bilateral bartering process of the Sugarscape model and verify it. We may come to the conclusion to collapse this paper into the fourth, validation paper if it is better suited there. In any case a paper will be written (I am constantly writing down insights, results and explanations during the process of research,

so writing up the paper in the end is only about restructuring and cleaning-up) whether it will be published or not is not that important - it will definitely go into the final thesis.

6.2.3 Category-Theory in ABS

In this paper we will present our research about a category-theoretical view on ABS. We may come to the conclusion to collapse this paper into the fourth, validation paper if it is better suited there. In any case a paper will be written (I am constantly writing down insights, results and explanations during the process of research, so writing up the paper in the end is only about restructuring and cleaning-up) whether it will be published or not is not that important - it will definitely go into the final thesis.

6.2.4 Validation in pure functional ABS

This paper will present our novel approach to validation in ABS using our FrABS library and category-theory. We will validate the decentralized bilateral bartering process as specified in the Sugarscape model against theoretical models and real-world examples. This paper is definitely intended to be a journal-paper because of its central importance to the PhD, its original novelty and the range and depth of the content. The target-journal is yet to be determined but we want to focus primarily on an agent-based modelling journal.

6.3 Years Overview

6.3.1 1st Year

In the first year, all is about orientation, experimenting and finding out what the PhD is *really* going to be about. In the remaining time the goal is to finalize the FrABS library and to release it on Hackage. We will start studying category-theory to build up knowledge to be used in the 3rd year.

6.3.2 2nd Year

In the second year, the focus will be on *verification*. Using the FrABS library and QuickCheck we will research how far we can go into formalizing model-specifications and how well we can do verification. The second focus we will be on continuing studying category-theory to gain a deep-enough understanding to use it for validation in the 3rd year.

6.3.3 3rd Year

In the third year I will look into *validation* with category-theory, focus on cleaning up the research and writing the final thesis. The plan is to start the writing

of the final thesis around April 2019 with a 5-months writing-window and to submit on-time at end of September 2019.

6.4 Conferences

- **Multi-Agent Systems AAMS** - General Multi-Agent Systems and Agent-Based Modelling & Simulation, Deadline in November
- **Social Simulation Conference SSC** - New Methods and models in simulation, Deadline in March
- **Symposium on Trends in Functional Programming** - Functional programming stuff, Deadline in May

6.5 Mile-Stones

- 2017 31st March - finished and submit Paper
- 2017 18th June - Finished writing 1st year report
- 2017 3rd July - Oral annual report
- 2017 October - 2nd year starts
- 2018 May - Submit Paper on FrABS
- 2018 October - 3rd year starts
- 2019 February - Submit Paper on Verification
- 2019 April - Begin of thesis-writing
- 2019 September - Submit Thesis
- 2019 30th September - official end of PhD
- 2020 30th September - end of pending-period

6.6 Fixed Holidays

- 2017 June 4th to 18th - 2 weeks on Amrum
- 2017 August 16th to 30th - 2 weeks in Austria
- 2017 22nd December to 7th January 2018 - 2 weeks Xmas
- 2018 22nd December 2018 to 6th January 2019 - 2 weeks Xmas

Chapter 7

Conclusions

7.1 Being Realistic

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.

7.2 What we are not doing

Because of this highly interdisciplinary topic we explicitly mention what we do not want to undertake in this PhD. First we don't want to develop another language for formal agent-specification which needs to be compiled or used in some fancy tool - we want to put it directly into Haskell, building on the existing facilities. Second, we are neither developing a new economic theory about decentralized bilateral bartering, we take the existing theory and existing agent-based models and apply our methods to them. Third, we don't want to use fancy statistics and number juggling for comparing validating and verifying models:

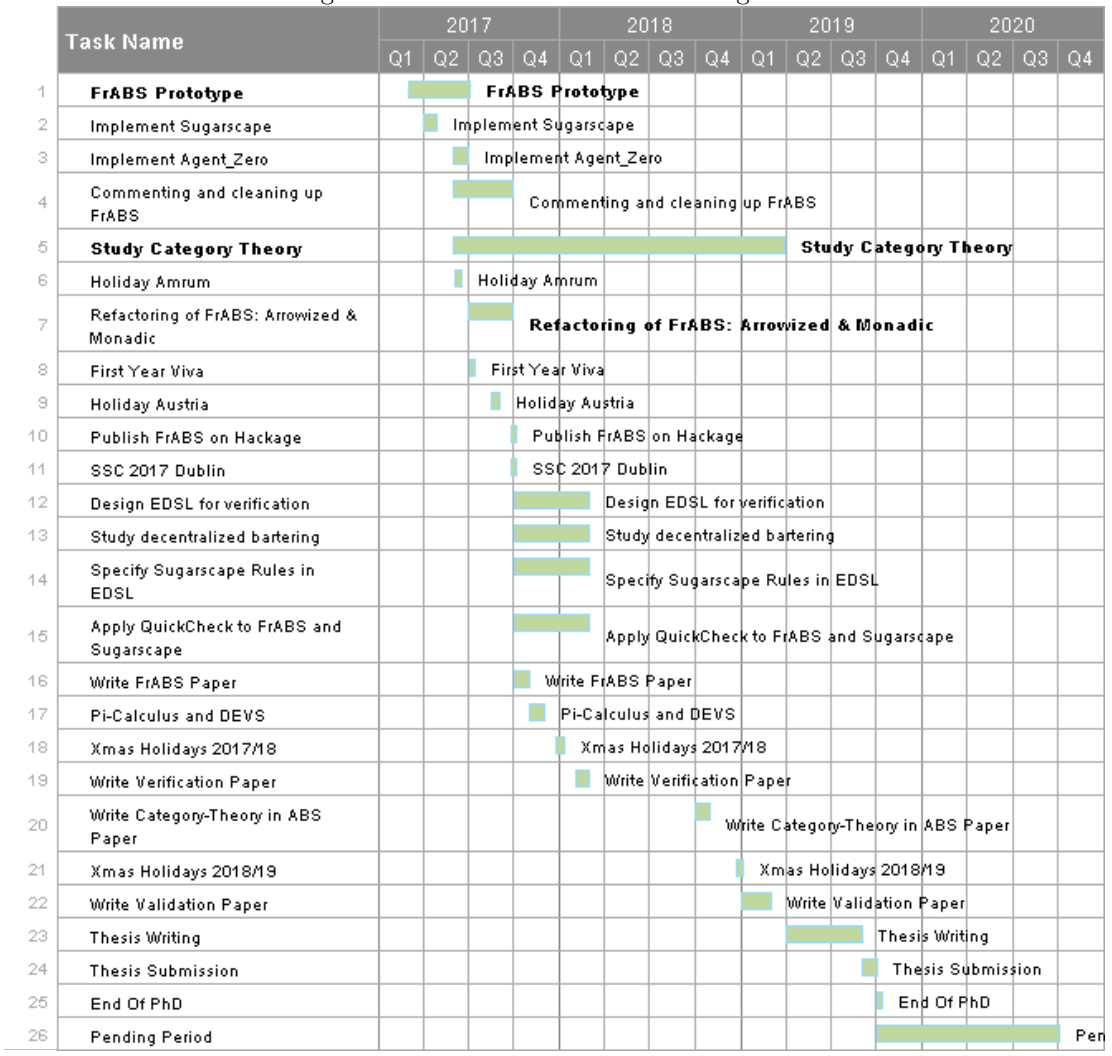
we want structural comparison (category-theory).

Appendices

Appendix A

Gantt-Chart

Figure A.1: Gantt-Chart for remaining PhD



Appendix B

Training Courses

- **Computer Science PGR Introductory Seminar** - 5 Dates
- **Tradition of Critique Lecture series** - Monday 29th September 2016 to Monday 8th December 2016 (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 - 23rd Feb 2017 (9:30 - 12:30)
- **Midland Graduate School 2017** - 9th - 13th April 2017 in Leicester, courses in Denotational Semantics, Naïve Type Theory and Testing with Theorem Provers.

Appendix C

Functional Reactive ABS (FrABS)

. In this chapter we present our approach to implementing ABS in the pure functional language Haskell and discuss the issues encountered [1]. As already described in our aims and objectives in Chapter 4, we are using the functional reactive programming (FRP) paradigm as implemented by the library Yampa and implement an ABS library on top of it [2]. When comparing our paradigms to the one of object-oriented Java we must solve fundamental problems in rather different ways.

1. Representing an agent and environment - there are no classes and objects in Haskell.
2. Interactions among agents and actions of agents on the environment - there are no method-calls and aliases in Haskell.
3. Implement the necessary update-strategies as discussed in our paper D, where we only focus on sequential- and parallel-strategies - there is no mutable data which can be changed implicitly through side-effects (e.g. the agents, the list of all the agents, the environment).

Before we can describe how we solved each of the problems, we first need to give an overview of the basic concepts of Yampa.

[1] This is not a real paper but only a basic introduction to our approach written for this 1st year report. Thus we assume knowledge of the concept of an agent and ABS already and wont explain these concepts again.

[2] We have implemented already a prototype together with the Sugarscape-, Agent_Zero, SIRS- and Schelling-Segregation Model which can be accessed from `https://github.com/thalerjonathan/phd/tree/master/coding/libraries/frABS/src`.

C.1 Yampa

The central concept of Yampa is the one of a signal-function which can be understood of a mapping from an input-signal to an output-signal. Whether the signal is discrete or continuous does not matter, Yampa is suited equally well to both kinds. Signal-functions are implemented in Yampa using continuations which allow to freeze program-state e.g. through closures and partial applications in functions which can be continued later:

```
type DTime = Double

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

Such a signal-function, which is called a *transition function* in Yampa, takes the amount of time which has passed since the previous time step and the current input signal (a). It returns a *continuation* of type SF a b determining the behaviour of the signal function on the next step and an output signal (b) of the current time-step.

Yampa provides a top-level function, running in the IO-Monad, which drives a signal-function by providing both input-values and time-deltas from callbacks. It is important to note that 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. We utilized this in the implementation of Recursive ABS (see Chapter 5).

Additional functionality which Yampa provides is the concept of Events which allow to implement changing behaviour of signal-functions at given points in time. An event can be understood to be similar to the Maybe-type of Haskell which either is an event with a given type or is simply NoEvent. Yampa provides facilities to detect if an event has fired and also provides functions to switch the signal-function into a new signal-function with changed behaviour. Another feature of Yampa is its EDSL for time-semantics: integration over time, delay, accumulation, holding, firing events after/now/repeatedly.

C.2 Agent Representation

An agent can be seen as a tuple $< id, s, m, ec, b >$.

- **id** - the unique identifier of the agent
- **s** - the generic state of the agent
- **m** - the set of messages the agent understands

- **ec** - the *type* of the environment-cells the agent may act upon
- **b** - the behaviour of the agent

C.2.1 Id

The id is simply represented as an Integer and must be unique for all currently existing agents in the system as it is used for message-delivery. A stronger requirement would be that the id of an agent is unique for the whole simulation-run and will never be reused - this would support replications and operations requiring unique agent-ids.

C.2.2 State

Each agent may have a generic state comprised of any data-type, most likely to be a structure.

```
data SIRSState = Susceptible | Infected | Recovered
data SIRSAgentState = SIRSAgentState {
    sirsState :: SIRSState,
    sirsCoord :: SIRSCoord,
    sirsTime :: Double
}
```

It is possible that the agent does not rely on any state s, then this will be represented by the unit type (). One wonders if this makes sense and asks how agents can then be distinguished between each other. In functional programming this is easily possible using currying and closures where one encapsulate initial state in the behaviour (see below), which allows to give each agent an individual initial state.

C.2.3 Messages

Agents communicate with each other through messages (see below) and thus need to have an agreed set of messages they understand. This is usually implemented as an ADT.

```
data SIRSMsg = Contact SIRSState
```

C.2.4 Environment-Cell

The agent needs to know the generic type of the cells the environment is made of to be able to act upon the environment. Note that at the moment we only implemented a discrete 2d environment and provide only access and manipulation to the cells in a 2d discrete fashion. In the case of a continuous n-dimensional environment this approach needs to be thoroughly revised. It is important to understand that it is the *type* of the cells and not the environment itself.

C.2.5 Behaviour

The behaviour of the agent is a signal-function which maps an AgentIn-Signal to an AgentOut-Signal. It has the following signature:

```
type AgentBehaviour s m e = SF (AgentIn s m e) (AgentOut s m e)
```

AgentIn provides the necessary data to the agent-behaviour: its id, incoming messages, the current state s, the environment (made out of the cells ec), its position in the environment and a random-number generator.

AgentOut allows the agent to communicate changes out of the behaviour: kill itself, create new agents, sending messages, state s, environment (made ouf of the cells ec), environment-position and random-number generator.

C.3 Environment

So far we only implemented a 2d-discrete environment. It can be understood to be a tuple of $< b, l, n, w, cs >$.

- **b** - the optional behaviour of the environment
- **l** - the limit of the environment: its maximum boundary extending from (0,0)
- **n** - the neighbourhood of the environment (e.g. Neumann, Moore, Manhattan...)
- **w** - the wrapping-type of the environment (clipping, horizontal, vertical, both)
- **cs** - the cells of the environment of type c

We represent the environment-behaviour as a signal-function as well but one which maps an environment to itself. It has the following signature:

```
type EnvironmentBehaviour c = SF (Environment c) (Environment c)
```

This is a regular SF thus having also the time of the simulation available and is called after all agents are updated. Note that the environment cannot send messages to agents because it is not an agent itself. An example of an environment behaviour would be to regrow some good on each cell according to some rate per time-unit (inspired by SugarScape regrowing of Sugar).

The cells are represented as a 2-dimensional array with indices from (0,0) to limit and a cell of type c at every position. Note that the cell-type c is the same environment-cell type ec of the agent.

Each agent has a copy of the environment passed in through the AgentIn and can change it by passing a changed version of the environment out through AgentOut.

C.4 Messaging

As discussed in the literature reflection in Chapter 3, inspired by the actor model we will resort to synchronized, reliable message passing with share nothing semantics to implement agent-agent interactions. Each Agent can send a message to an other agent through AgentOut-Signal where the messages are queued in the AgentIn-Signal and can be processed when the agent is updated the next time. The agent is free to ignore the messages and if it does not process them they will be simply lost. Note that due to the fact we don't have method-calls in FP, messaging will always take some time, which depends on the sampling interval of the system. This was not obviously clear when implementing ABS in an object-oriented way because there we can communicate through method calls which are a way of interaction which takes no simulation-time.

C.5 Conversations

The messaging as implemented above works well for one-directional, virtual asynchronous interaction where we don't need a reply at the same time. A perfect use-case for messaging is making contact with neighbours in the SIRS-model: the agent sends the contact message but does not need any response from the receiver, the receiver handles the message and may get infected but does not need to communicate this back to the sender. A different case is when agents need to transact in the time-step one or multiple times: agent A interacts with agent B where the semantics of the model (and thus messaging) need an immediate response from agent B - which can lead to further interactions initiated by agent A. The Sugarscape model has three use-cases for this: sex, warfare and trading amongst agents all need an immediate response (e.g. wanna mate with me?, I just killed you, wanna trade for this price?). The reason is that we need to transact now as all of the actions only work on a 1:1 relationship and could violate ressource-constraints. For this we introduce the concept of a conversation between agents. This allows an agent A to initiate a conversation with another agent B in which the simulation is virtually halted and both can exchange an arbitrary number of messages through calling and responding without time passing (something not possible without this concept because in each iteration the time advances). After either one agent has finished with the conversation it will terminate it and the simulation will continue with the updated agents (note the importance here: *both* agents can change their state in a conversation). The conversation-concept is implemented at the moment in the way that the initiating agent A has all the freedom in sending messages, starting a new conversation,... but that the receiving agent B is only able to change its state but is not allowed to send messages or start conversations in this process. Technically speaking: agent A can manipulate an AgentOut whereas agent B can only manipulate its next AgentIn. When looking at conversations they may look like an emulation of method-calls but they are more powerful: a receiver can be unavailable to conversations or simply refuse to handle this conversation.

This follows the concept of an active actor which can decide what happens with the incoming interaction-request, instead of the passive object which cannot decide whether the method-call is really executed or not.

C.6 Iteration-Strategies

Building on the foundations laid out in my paper about iteration-strategies in Appendix D, we implement two of the four strategies: sequential- and parallel-strategy. We deliberately ignore the concurrent- and actor-strategy for now and leave this for further research [3]. Implementing iteration-strategies using Haskell and FRP is not as straight-forward as in e.g. Java because one does not have mutable data which can be updated in-place. Although my work on programming paradigms in Appendix E did not take FRP into account, general concepts apply equally as well.

C.6.1 Sequential

In this strategy the agents are updated one after another where the changes (messages sent, environment changed,...) of one agent are visible to agents updated after. Basically this strategy is implemented as a variant of fold which allows to feed output of one agent (e.g. messages and the environment) forward to the other agents while iterating over the list of agents. For each agent the agent-behaviour signal-function is called with the current AgentIn as input to retrieve the according AgentOut. The messages of the AgentOut are then distributed to the receivers AgentIn. The environment of the agent, which is passed in through AgentIn and returned through AgentOut will then be passed forward to all agents i + 1 AgentIn in the current iteration and override their old environment. Thus all steps of changes made to the environment are visible in the AgentOuts. The last environment is then the final environment in the current iteration and will be returned by the callback function together with the current AgentOuts.

C.6.2 Parallel

The parallel strategy is *much* easier to implement than the sequential but is of course not applicable to all models because of it different semantics. Basically this strategy is implemented as a map over all agents which calls each agent-behaviour signal-function with the agents AgentIn to retrieve the new AgentOut. Then the messages are distributed amongst all agents. A problem in this strategy is that the environment is duplicated to each agent and then each agent can work on it and return a changed environment. Thus after one iteration there are n versions of environments where n is equals to the number

[3] Also both strategies would require running in the STM-Monad, which is not possible with Yampa. The work of Ivan Perez in [**?**] implemented a library called Dunai, which is the same as Yampa but capable of running in an arbitrary Monad.

of agents. These environments must then be collapsed into a final one which is always domain-specific thus needs to be done through a function provided in the environment itself.

C.7 Further Research

In his 1st year report about Functional Reactive GUI programming, Ivan Perez [4] writes: "FRP 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". This of course raises the question whether FRP is *really* the right approach, because the way we implement ABS, message-passing is an essential concept. It is important to emphasis that agent-relations in interactions are never fixed in advance and are completely dynamic, forming a network. Maybe one has to look at message passing in a different way in FRP, and to view and model it as a data-dependency but it is not clear how this can be done. The question is whether there is 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 concept of conversations (see above) are a way to go but we leave this for further research at the moment.

[4] main author of the paper [**?**]

Appendix D

Update-Strategies

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

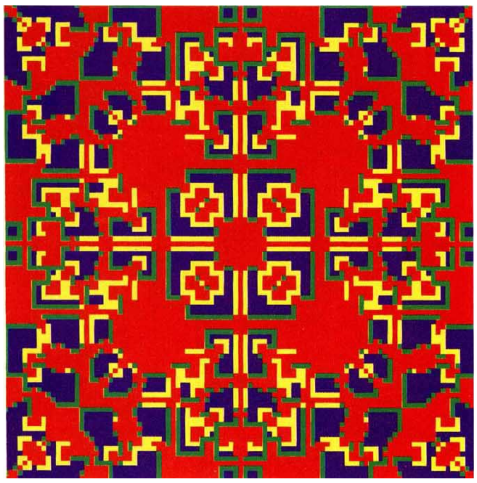


Figure 1: Patterns formed by playing the *Prisoners Dilemma* game on a 99x99 grid with $1.8 < b < 2$ after 217 steps with all agents being cooperators except one defector at the center. Blue are cooperators, red are defectors, yellow are cooperators which were defectors in the previous step, green are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

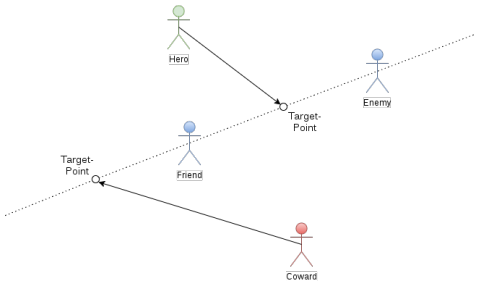


Figure 2: A conceptual diagram of the *Heroes & Cowards* game. Hero (green) and coward (red) have the same agents as friend and enemy but act different: the hero tries to move in between the friend and enemy whereas the coward tries to hide behind its friend.

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.

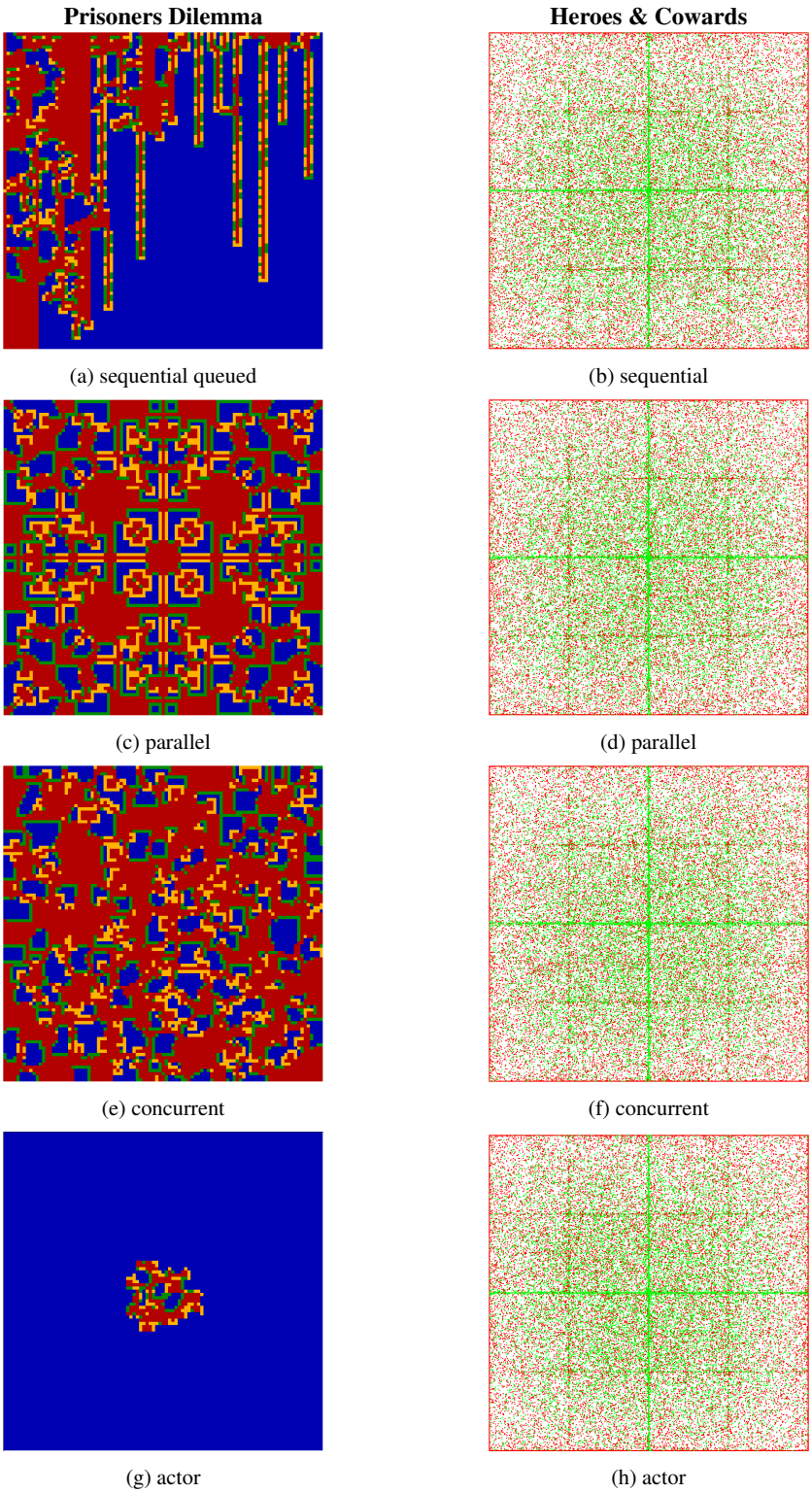
3.3.3. Summary. To conclude, the most natural and common update-strategy in these toolkits is the *sequential strategy* which is not very surprising. The primary target are mostly agent-based modellers which are non-computer scientists so the toolkits also try to be as simple as possible and multi-threading 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 E

Programming-paradigms in ABS

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 F

Questions & Answers

In this chapter I give answers to anticipated questions and objections about my research direction and vision of doing pure functional ABS [1].

Doesn't NetLogo provide all this? Fair point, NetLogo can be seen as a functional language. Problem: side-effects, global state, dynamic, very difficult to verify. Main benefits: quick and easy to learn, for beginners.

Why another formalism, don't we have already enough of them? It is only partly another formalism - the important fact is, that it is directly built into Haskell, thus leveraging on a pure functional programming language directly than having some formal language which is then translated to machine code.

Pure functional programming? You must be kidding! Objects are so close to Agents! Exactly this is the problem: objects are in fact very close to agents but they have also very important differences which are quickly cast aside when implementing ABS in e.g. Java leading to problems like shared state.

What about parallelism and concurrency? Although very important, they are not of primary interest in this PhD. It is very well known that pure functional programming is exceptionally well suited in implementing parallel programs (no side-effects) and also concurrent ones (through STM, allowing composition of concurrency) and we have looked into this a bit in the paper on programming paradigms in ABS, so all of this can be applied to our research as well.

[1] They are not always posed in a dead-serious way but as it is a quite controversial topic - ABS should be done object-orientated after all huh? - I think it is appropriate. Also some objections were raised in exactly this way.

You mentioned performance and space-leaks as weaknesses? Yes this is probably the largest issue to date as even problems with small number of agents (500-1000) are getting very slow. Although performance is not of interest here we are well aware that slow software is simply not used, thus we are looking for ways in improving the speed.

You mentioned the difficulty of debugging in weaknesses of pure functional programming, how can you be sure your program is correct? Good point but having the ability to step through a program alone does not guarantee the correctness of a program. In Haskell the static type system enforces already so much and prevents bugs which are normal in e.g. Java, also the lack of implicit side-effects makes programs much less prone to errors. In the end there is also reasoning techniques and QuickCheck. Thus we have much more power at hand in detecting bugs and ensuring correctness than classic oop languages like Java or C++.

So you say ABS should be done in Haskell using your library and using your techniques and object-orientation sucks? You're mixing things up but: it depends. If you are familiar with Haskell and like functional programming, go for it. If you are an oop-disciple, who can code the most complex ABS just don't. If you need correctness and verification in your simulation then definitely go for it.

Nobody has done it before? Probably for a good reason, object-oriented is the way to go for ABS! At least to my knowledge, backed-up by a thorough literature-review, no one has done a proper treatment of ABS in a pure functional way. My original motivation was just one thing: curiosity! No one has done it and I want to find out how it can be done, what its benefits are, what its weaknesses are. Just to claim oo is the *right* way is not a serious claim unless you haven't tried other ways and compared them. I have done both ways in-depth, have you?

What are the benefits? Simulations are more likely to be correct, better support for verification, no gap between specification and code, easier to replicate.

What are the weaknesses? Performance, steep learning curve if you don't know functional programming.

Why not using Erlang? Its functional, it has the actor model built in? Yes, Erlang would be nearly perfect but the non-determinism and asynchronous nature of the messaging is the main problem which can hardly be overcome. If you don't need determinism (as in every run may have different dynamics despite using the same RNG) in your model, then Erlang (or Scala) is definitely an interesting option.

But why functional? Is this unique to functional? All this can be done in principle in other languages! In principle you can implement an ABS in assembly, machine code, lambda calculus or even a Turing machine. The point is that the languages in which we program shape and define how we think of a problem and pure functional programming makes you think in a much more high level and abstract way which is much more suitable to validation and verification. Also it allows to draw parallels and map concepts from category theory (yes yes of course you could do that in principle in all the above cited languages. Also in principle I could drive in a wheelchair from Chile to Alaska).