Functional Reactive Agent-Based Simulation

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

Thorsten Altenkirch
School of Computer Science
University of Nottingham
thorsten.altenkirch@nottingham.ac.uk

***Abstract*—Agent-Based Simulation (ABS) is a methodology in which a system is simulated in a bottom-up approach by modelling the (micro) interactions of its constituting parts, called agents, out of which interactions the global (macro) system behaviour emerges. So far, the Haskell community hasn't been much in contact with the community of ABS due to the latter's primary focus on the object-oriented programming paradigm. This paper tries to bridge the gap between those two communities by introducing the Haskell community to the concepts of ABS using the simple SIR model from epidemiology. Further we present our library *FrABS* which allows to implement ABS the first time in Haskell. In this library we leverage the basic concepts of ABS with functional reactive programming using Yampa which results in a surprisingly powerful and convenient EDSL for formulating ABS.**

***Index Terms*—Functional Reactive Programming, Agent-Based Simulation**

I. Introduction

In Agent-Based Simulation (ABS) one models and simulates a system by modeling and implementing the pro-active constituting parts of the system, called *Agents* and their local interactions. From these local interactions then the emergent property of the system emerges. ABS is still a young field, having emerged in the early-to-mid 90s primarily in the fields of social simulation and computational economics.

The authors of the seminal Sugarscape model [1] explicitly advocate object-oriented programming as "a particularly natural development environment for Sugarscape specifically and artificial societies generally.". They implemented their simulation software in Object Pascal and C where they used the former for programming the agents and the latter for low-level graphics [2]. Axelrod [3] recommends Java for experienced programmers and Visual Basic for beginners. Up until now most of the ABS community seems to have followed these suggestions and are implemented using programming languages of the object-oriented imperative paradigm.

A 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 [4] - it is passive. Also it is remarkable that [4] 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 [5] 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 [6] 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 [3]. As ABS is almost always used for scientific research, producing often break-through scientific results as pointed out in [5] and used for policy making, these ABS need to be *free of bugs*, *verified against their specification*, *validated against hypotheses* and ultimately be *reproducible*.

Pure functional programming in Haskell claims [7], [8] to overcome these problems or at least allows to tackle them more effectively due to its declarative nature, free of side-effects, strong static type system. In this paper we ask how ABS can be done in Haskell, what the benefits are and if it could overcome the issues mentioned above.

II. Agent Based Simulation

We understand ABS as a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is of knowledge. Those parts, called agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. So the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in a network of neighbours by exchange of messages [9]. 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 over which they have full, exclusive control.

- They are pro-active which means they can initiate actions on their own e.g. change their internal state, send messages, create new agents, terminate themselves.
- They are situated in an environment and can interact with it.
- They can interact with other agents which are situated in the same environment by means of message-passing.

Epstein [10] identifies ABS to be especially applicable for analysing *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. Thus in the line of the simulation models *Statistical* †, *Markov* ‡, *System Dynamics* §, *Discrete Event* ∓, ABS is the most powerful one as it allows to model the following:

- Linearity & Non-Linearity †‡§∓ - the dynamics of the simulation can exhibit both linear and non-linear behaviour.
- Time †‡§∓ - agents act over time, time is also the source of pro-activity.
- States ‡§∓ - agents encapsulate some state which can be accessed and changed during the simulation.
- Feedback-Loops §∓ - because agents act continuously and their actions influence each other and themselves, feedback-loops are the norm in ABS.
- Heterogeneity ∓ - although agents can have same properties like height, sex,... the actual values can vary arbitrarily between agents.
- Interactions - agents can be modelled after interactions with an environment or other agents, making this a unique feature of ABS, not possible in the other simulation models.
- Spatiality & Networks - agents can be situated within e.g. a spatial (discrete 2d, continuous 3d,...) or network environment, making this also a unique feature of ABS, not possible in the other simulation models.

III. The SIR Model

To explain the concepts of ABS and of our functional reactive approach to it, we introduce the SIR model as a motivating example. The SIR model is a a very well studied and understood compartment model from epidemiology which allows to simulate the dynamics of an infectious disease spreading through a population. In this model, people in a population of size N can be in either one of three states *Susceptible*, *Infected* or *Recovered* at a particular time, where it is assumed that initially there is at least one infected person in the population. People interact with each other *on average* with a given rate β per time-unit and get infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time-units and is then immune to further infections. An infected person interaction with another infected one is never re-infected, thus interactions amongst infected people is not important in this model. This definition gives rise to three compartments with the transitions as seen in Figure 1.

The dynamics of this model over time can be formalized using the following equations:

Fig. 1: Transitions in the SIR compartment model.

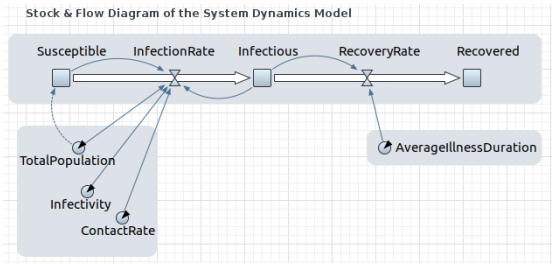


Fig. 2: A visual representation of the stocks and flows of the SIR compartment model in the AnyLogic Software.

$$\begin{aligned}\frac{\mathrm{d}S}{\mathrm{d}t} &= -infectionRate\\ \frac{\mathrm{d}I}{\mathrm{d}t} &= infectionRate - recoveryRate\\ \frac{\mathrm{d}R}{\mathrm{d}t} &= recoveryRate\end{aligned}$$

$$\begin{aligned}infectionRate &= \frac{I\beta S\gamma}{N}\\ recoveryRate &= \frac{I}{\delta}\end{aligned}$$

Solving these can be done using the System-Dynamics (SD) approach which solves the equations by integrating over time. In the SD terminology, the integrals are called *Stocks* and the values over which is integrated over time are called *Flows* [1].

$$\begin{aligned}S(t) &= N + \int_0^t -infectionRate\,\mathrm{d}t\\ I(t) &= 1 + \int_0^t infectionRate - recoveryRate\,\mathrm{d}t\\ R(t) &= \int_0^t recoveryRate\,\mathrm{d}t\end{aligned}$$

There exist a huge number of software-packages which allow to conveniently express SD models using a visual approach like in Figure 2.

Running the SD simulation over time results in the dynamics as shown in Figure 3 with the given variables.

A. An Agent-Based approach

The SD approach is inherently Top-Down because the emergent property of the system is formalized in differential equations. The question is if such a top-down behaviour can be emulated using ABS, which is inherently bottom-up. Also the question is if there are fundamental drawbacks and benefits when doing so using ABS. Indeed such questions were asked before and modelling the SD approach of the SIR model is possible using an agent-based approach. It is important to note that SD treats the population completely continuous which results in non-discrete values of stocks e.g. 3.1415 infected persons. Thus the fundamental approach to map the SIR model to an ABS is to discretisize the population and model each person in the population as an individual agent. The transition between the states are no longer happening according to

[1]The $+1$ in $I(t)$ amounts to the initially infected agent - if there wouldn't be a single infected one, the system would immediately reach equilibrium.

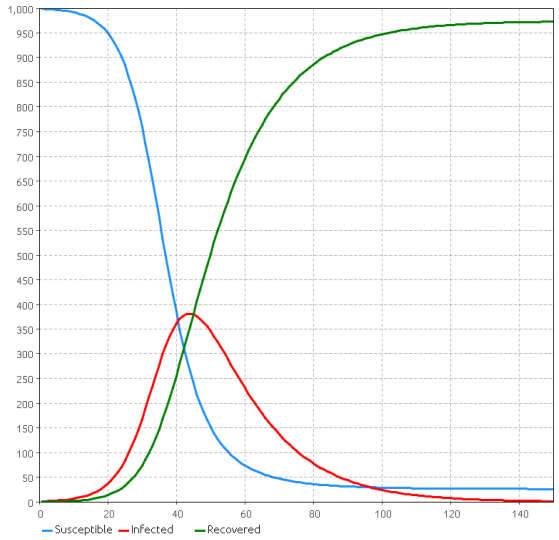


Fig. 3: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size $N = 1000$, contact rate $\beta = 1/5$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$. 1 initially infected agent. Simulation run of 150 time-steps. Dynamics generated with AnyLogic Personal Learning Edition 8.1.0.

continuous differential equations but due to discrete events caused both by interactions amongst the agents and time-outs. The behaviour can be defined as follows:

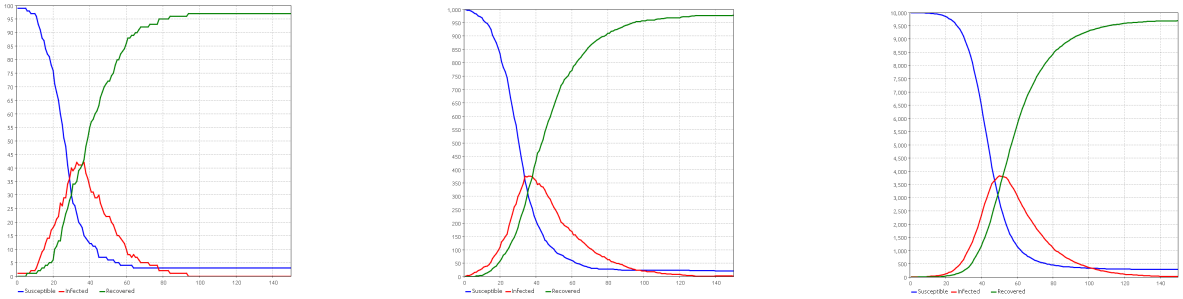
- Every agent makes on average contact with β random other agents per time unit. In ABS we can only contact discrete agents thus we model this by generating a random event on average every β time units (Note that this amounts to sampling from an exponential CDF).
- An agent does not know the other agents' state when making contact with it, thus we need a mechanism in which agents reveal their state in which they are in *at the moment of making contact*. Obviously the already mentioned messaging-mechanism which allows agents to interact is perfectly suited to do this.
  - *Susceptible* agent: sends a "Susceptible" message when contacting another agent. There is no need to reply to other incoming messages as making contact with a susceptible agent has no influence on the state of an agent.
  - *Infected* agent: An infected agent needs to reply to incoming "Susceptible" messages with an "Infected" message to let the susceptible agent know that it has made contact with an infected agent. Note that an infected agent must not sends a "Infected" message by initiated on its own as this would lead to false results (TODO: better argumentation).
  - *Recovered* agent: does not need to send messages because contacting it or being contacted by it has no influence on the state.
- Susceptible to Infected: needs to have made contact with an infected agent which happens when it receives an "Infected" message. If this occurs an infection occurs with a probability of γ. The infection can be calculated by drawing from a uniform random-distribution between 0 and 1 and comparing the value to γ, if the drawn value $p < \gamma$ then infection occurs. Note that this needs to be done for *every* received "Infected" message.
- Infected to Recovered: a person recovers *on average* after δ time unites. This is implemented by drawing the duration from an exponential distribution (TODO: borchschev) with $\lambda = \frac{1}{\delta}$ and making the transition after this duration.

We will discuss the implementation of this approach in the following sections and as will be shown FrABS will allow us to express this behaviour very explicitly looking very much like a formal ABS specification of the problem. In Figure 4 we give the dynamics simulating the SIR model with the agent-based approach.

As previously mentioned the agent-based approach is a rather discrete one which means that with increasing number of agents, the discrete dynamics aproximate the continuous dynamics of the SD simulation. Still the dynamics of 10,000 Agents do not match the dynamics of the SD simulation to a satisfactory level as the Susceptibles in SD fall off earlier and the peak is reached a bit earlier. This is because as opposed to the SD simulation the agent-based approach is inherently a stochastic one as we continuously draw from random-distributions which drive our state-transitions. What we see in Figure 4 is then just a single run where the dynamics would result in slightly different shapes when run with a different random-number generator seed. The agent-based approach thus generates a distribution of dynamics over which ones needs to average to arrive at the correct solution. This can be done using replications in which the simulation is run with the exact same parameters multiple times but each with a different random-number generator see. The resulting dynamics are then averaged and the result is then regarded as the correct dynamics. We have done this as can be seen in Figure 5, using 1000 replications, which now completely matches the SD dynamics [2].

For a more in-depth discussion of how to approximate an SD model by ABS see [11] which explain that we need to draw the illness-duration from an exponential-distribution

[2]Note that in the replications we are using 10 initially infected agents to ensure that no simulation run will terminate too early (meaning that the disease gets extinct after a few time steps) which would offset the dynamics completely. This happens due to "unlucky" random distributions which can be repaired by introducing more initially infected agents which increases the probability of spreading the disease in the very early stage of the simulation drastically. We found that when using 10 initially infected agents in a population of 10,000 (which amounts to (0.1%)) is enough to never result in an early terminating simulation. This is also a fundamental difference between SD and ABS: the dynamics of the agent-based approach can result in a wide range of scenarios which includes also the one that the disease gets extinct in the early stages (a lucky coincidence for mankind) - this is simply not possible in the SD approach. So we can argue that ABS is much closer to reality than SD as it allows to explore alternate futures in the dynamics.



(a) 100 Agents (b) 1000 Agents (c) 10,000 Agents

Fig. 4: Approximating the continuous dynamics of the SD simulation using the agent-based approach. Model-parameters are the same ($\beta = 1/5$, $\gamma = 0.05$, $\delta = 15$, 1 initially infected agent) except population size. All Simulations run for 150 time-steps. Dynamics generated with AnyLogic Personal Learning Edition 8.1.0.

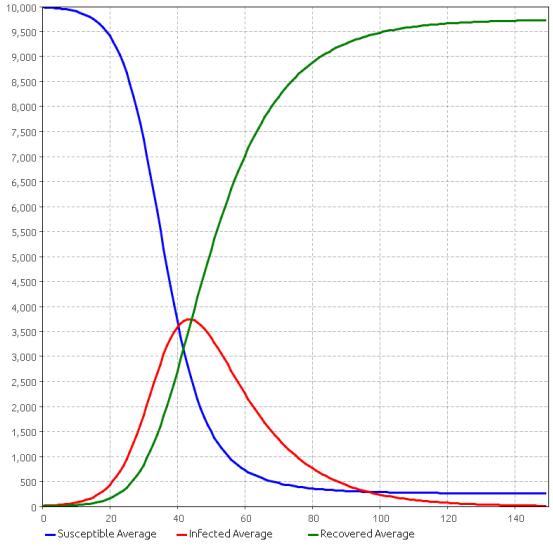


Fig. 5: Dynamics of the SIR compartment model using the agent-based approach with same parameters as in SD ($\beta = 1/5$, $\gamma = 0.05$, $\delta = 15$) but with a population of 10,000 where 10 are initially infected and the dynamics averaged over 1000 replications. Dynamics generated with AnyLogic Personal Learning Edition 8.1.0.

because the illness-duration is proportional to the size of the infected. note: this is wrongly expressed, need to find the correct formulation. Also [12]

IV. The formal Agent-Model

TODO: what are the difficulties when implementing an ABS?

TODO: cite my own work on update-strategies

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

- **id** - the unique identifier of the agent
- **s** - the generic state of the agent
- **m** - the set of messages the agent understands
- **e** - the *type* of the environment-cells the agent may act upon
- **b** - the behaviour of the agent

A. 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.

B. 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. 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
```

D. 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.

E. 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.

V. Environment

TODO: update, we have now also 2d-continuous AND networks!

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

- **b** - the optional behaviour of the environment
- **d** - the dimensions of the environment: its maximum boundary extending from (0,0)
- **n** - the neighbourhood of the environment (Neumann, Moore)
- **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.

VI. Functional Reactive ABS

the fundamental problem is that unlike in oo e.g. java there are no objects and no implicit aliases through which to acess and change data: method calls are not there in FP. we must silve the problem of how to represent an agent and how agents can interact with each other

using example SIRS or Wildfire TODO: show how simple state-transition agents work using switch TODO: show how the time-semantics can be used

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 **??**, 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).

A. Messaging

As discussed in the literature reflection in Chapter **??**, 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.

B. 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. Iteration-Strategies

Building on the foundations laid out in my paper about iteration-strategies in Appendix **??**, 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 **??** did not take FRP into account, general concepts apply equally as well.

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.

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 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.

D. Environment

TODO: again cite my own work where I discussed the problem of environments

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. In the sequential update-strategy the environment of the agent i will then be passed 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 int he current iteration and will be returned by the callback function together with the current AgentOuts. In the parallel update-strategy the environment is duplicated to each agent and then each agent can work on it and return the changed environment. Thus after one iteration there are n versions of environments where n is equals to the number 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. In both the sequential and parallel update-strategy after one iteration there is one single environment left. An environment can have an optional behaviour which allows the environment to update its cells. This is a regular SF thus having also the time of the simulation available. 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).

E. Time-Semantics

The main reason for building our pure functional ABMS approach on top of Yampa was to leverage the powerful time-semantics of Yampa which allows us to implement important concepts of ABMS:

state-chart: agents are at all time of their life-cycle in one state and can switch between multiple states using transitions timed transitions: transition to another state/behaviour happens at a discrete time rate transitions: transition happens with a

[3]Also both strategies would require running in the STM-Monad, which is not possible with Yampa. The work of Ivan Perez in [13] implemented a library called Dunai, which is the same as Yampa but capable of running in an arbitrary Monad.

given rate message transition: transition upon receiving a given message

F. Agents as Signals

Due to the underlying nature and motivation of Functional Reactive Programming (und im speziellen) Yampa, Agents can be seen as Signals which is generated and consumed by a Signal-Function which is the behaviour of an Agent. If an Agent does not change the OUTPUT-signal is constant, if the agent changes e.g. by sending a message, changing its state,... the OUTPUT signal changes. A dead agent has no signal at all.

G. Time-Sampling

sampling rate depends on the transition times & rates of the model. when e.g. the contact rate is 5 then the sampling dt should be below 0.2

H. System Dynamics

can emulate system dynamics due to the parallel update-strategy and continuous time-flow semantics

I. Discrete Event Simulation

DES in FrABMS? how easily can we implement server/queue systems? do they also look like a specification? potential problem: ordering of messages is not guaranteed by now

VII. Discussion

advantages: - no side-effects within agents leads to much safer code - edsl for time-semantics - declarative style: agent-implementation looks like a model-specification - reasoning and verification - sequential and parallel - powerful time-semantics - arrowized programming is optional and only required when utilizing yampas time-semantics. if the model does not rely on time-semantics, it can use monadic-programming by building on the existing monadic functions in the EDSL which allow to run in the State-Monad which simplifies things very much - when to use yampas arrowized programing: time-semantics, simple state-chart agents - when not using yampas facilities: in all the other cases e.g. SugarScape is such a case as it proceeds in unit time-steps and all agents act in every time-step - can implement System Dynamics building on Yampas facilities with total ease - get replications for free without having to worry about side-effects and can even run them in parallel without headaches - cant mess around with time because delta-time is hidden from you (intentional design-decision by Yampa). this would be only very difficult and cumbersome to achieve in an object-oriented approach. TODO: experiment with it in Java - how could we actually implement this? I think it is impossible: may only achieve this through complicated application of patterns and inheritance but then has the problem of how to update the dt and more important how to deal with functions like integral which accumulates a value through closures and continuations. We could do this in OO by having a general base-class e.g. ContinuousTime which provides functions like updateDt and integrate, but we could only accumulate a single integral value. - reproducibility statically guaranteed - cannot mess around with dt - code == specification - rule out serious class of bugs - different time-sampling leads to different results e.g. in wildfire & SIR but not in Prisoners Dilemma. why? probabilistic time-sampling? - reasoning about equivalence between SD and ABS implementation in the same framework - recursive implementations

- we can statically guarantee the reproducibility of the simulation because: no side effects possible within the agents which would result in differences between same runs (e.g. file access, networking, threading), also timedeltas are fixed and do not depend on rendering performance or userinput

disadvantages: - performance is low - reasoning about performance is very difficult - very steep learning curve for non-functional programmers - learning a new EDSL - think ABMS different: when to use async messages, when to use sync conversations

[ ] important: increasing sampling freqzency and increasing number of steps so that the same number of simulation steps are executed should lead to same results. but it doesnt. why? [ ] hypothesis: if time-semantics are involved then event ordering becomes relevant for emergent patterns. there are no tine semantics in heroes and cowards but in the prisoners dilemma [ ] can we implement different types of agents interacting with each other in the same simulation ? with different behaviour funcs, digferent state? yes, also not possible in NetLogo to my knowledge. but they must have the same messages, emvironment

[ ] Hypothesis: we can combine with FrABS agent-based simulation and system dynamics

VIII. Related Research

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 [14], [15], [16]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [17]. It comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. [16] which discuss using functional programming for DES mention the paradigm of functional reactive programming (FRP) to be very suitable to DES. 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 [18]. Although the fields of games and ABS seem to be very different, in the end they have also very important similarities: both are simulations which perform numerical computations and update objects in a loop either concurrently or sequential [4]. In games these objects are called *game-objects* and in ABS they are called *agents* but they are conceptually the same thing. The two main points

[4]Gregory [19] defines computer-games as *soft real-time interactive agent-based computer simulations*

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 [20] 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.

[21] and [22] 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.

IX. Conclusion and future research

TODO: don't sell this paper as an opposing view against OOP (e.g. OOP is bad) but as a positive view: "for the first time it is possible to do ABMS in pure functional programming".

TODO: the first (of two) contribution of this paper is: an explanation of one way of how ABS can be done in pure functional programming and its benefits: declarative style where the code looks very much like specification, fewer LoC, fewer Bugs, reasoning and proves

TODO: main second contribution is: show that the SD and the ABS implementation of the SIR model are the same by proving that ABS solves the SD equation. this should be possible by now using reasoning techniques (and quickcheck?)

further research - verification & validation - switch to Dunai to allow usage of Monadic programming in the arrows - use dunai to implement concurrent & actor strategy by running in the STM monad

In his 1st year report about Functional Reactive GUI programming, Ivan Perez [5] 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.

future research: STM in FrABS: run them in parallel but concurrently and advance time through a separate STM loop in the main loop. constant time should then keep the agents constant unless some discrete event happens e.g. message arrives. sugarscape wouldnt work but FrSIR. but still we have random results. also i could implement this in java unless i use STM specific stuff

random local time-steps: we can emulate the behaviour of actors but with reproducible results by using dunai and letting each agent do its own time-sampling with random intervals in a given range: should use parallel strategy

[5] main author of the paper [13]

Acknowledgments

The authors would like to thank I. Perez, H. Nilsson and S. Venkatesan for constructive comments and valuable discussions.

References

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

[2] R. Axtell, R. Axelrod, J. M. Epstein, and M. D. Cohen, "Aligning simulation models: A case study and results," *Computational & Mathematical Organization Theory*, vol. 1, no. 2, pp. 123–141, Feb. 1996. [Online]. Available: https://link.springer.com/article/10.1007/BF01299065

[3] R. Axelrod, "Advancing the Art of Simulation in the Social Sciences," in *Simulating Social Phenomena*. Springer, Berlin, Heidelberg, 1997, pp. 21–40, dOI: 10.1007/978-3-662-03366-1_2. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-662-03366-1_2

[4] N. R. Jennings, "On Agent-based Software Engineering," *Artif. Intell.*, vol. 117, no. 2, pp. 277–296, Mar. 2000. [Online]. Available: http://dx.doi.org/10.1016/S0004-3702(99)00107-1

[5] R. Axelrod, "Chapter 33 Agent-based Modeling as a Bridge Between Disciplines," in *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed. Elsevier, 2006, vol. 2, pp. 1565–1584, dOI: 10.1016/S1574-0021(05)02033-2. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574002105020332

[6] ——, "The Convergence and Stability of Cultures: Local Convergence and Global Polarization," Santa Fe Institute, Working Paper, Mar. 1995. [Online]. Available: http://econpapers.repec.org/paper/wopsafiwp/95-03-028.htm

[7] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A History of Haskell: Being Lazy with Class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–1–12–55. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238856

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

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

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

[11] A. Borshchev and A. Filippov, "From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools," Oxford, Jul. 2004.

[12] C. M. Macal, "To Agent-based Simulation from System Dynamics," in *Proceedings of the Winter Simulation Conference*, ser. WSC '10. Baltimore, Maryland: Winter Simulation Conference, 2010, pp. 371–382. [Online]. Available: http://dl.acm.org/citation.cfm?id=2433508.2433551

[13] I. Perez, M. Brenz, and H. Nilsson, "Functional Reactive Programming, Refactored," in *Proceedings of the 9th International Symposium on Haskell*, ser. Haskell 2016. New York, NY, USA: ACM, 2016, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2976002.2976010

[14] T. De Jong, "Suitability of Haskell for Multi-Agent Systems," University of Twente, Tech. Rep., 2014.

[15] M. Sulzmann and E. Lam, "Specifying and Controlling Agents in Haskell," Tech. Rep., 2007.

[16] P. Jankovic and O. Such, "Functional Programming and Discrete Simulation," Tech. Rep., 2007.

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

[18] T. Sweeney, "The Next Mainstream Programming Language: A Game Developer's Perspective," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 269–269. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111061

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

[20] N. Bezirgiannis, "Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism," Ph.D. dissertation, Utrecht University - Dept. of Information and Computing Sciences, 2013.

[21] O. Schneider, C. Dutchyn, and N. Osgood, "Towards Frabjous: A Two-level System for Functional Reactive Agent-based Epidemic Simulation," in *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium*, ser. IHI '12. New York, NY, USA: ACM, 2012, pp. 785–790. [Online]. Available: http://doi.acm.org/10.1145/2110363.2110458

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