First contact: Agents meet Haskell

Simulating epidemics using Functional Reactive Programming

An Agent-Based Approach

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

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

Abstract

TODO: cite my own 1st paper from SSC2017: add it to citations
TODO: refine it: start with simulating epidemics and then go into ABS
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 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. We do this by deriving an agent-based implementation for the simple SIR model from epidemiology. In our approach we leverage the basic concepts of ABS with functional reactive programming from Yampa and Dunai which results in a surprisingly fresh, powerful and convenient EDSL for programming ABS in Haskell.

Index Terms

Functional Reactive Programming, Agent-Based Simulation

I. Introduction

In this paper we derive a pure functional approach for agent-based simulation in Haskell. We start from a very simple solution running in the Random Monad, then making the transition to Yampa

The aim of this paper is to show how ABS can be done in Haskell and what the benefits and drawbacks are. We do this by introducing the SIR model of epidemiology and derive an agent-based implementation for it based on Functional Reactive Programming. By doing this we give the reader a good understanding of what ABS is, what the challenges are when implementing it and how we solved these in our approach. We then discuss details which must be paid attention to in our approach and its benefits and drawbacks. The contribution is a novel approach to implementing ABS with powerful time-semantics and more emphasis on specification and possibilities to reason about the correctness of the simulation.

II. Defining Agent-Based Simulation

Agent-Based Simulation (ABS) is a methodology to model and simulate 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 some neighbourhood by exchange of messages [1]. We informally assume the following about our agents TODO: need some references here, we cannot claim this without citation here (cite Peers book):

- 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 messaging.

Epstein [2] 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 types *Statistic* †, *Markov* ‡, *System Dynamics* §, *Discrete Event* $^{\mp}$, ABS is the most powerful one as it allows to model the following:

- Linearity & Non-Linearity $^{\dagger\ddagger\S\mp}$ - the dynamics of the simulation can exhibit both linear and non-linear behaviour.
- Time $^{\dagger\ddagger\S\mp}$ - agents act over time, time is also the source of pro-activity.
- States $^{\ddagger\S\mp}$ - agents encapsulate some state which can be accessed and changed during the simulation.

Fig. 1: Transitions in the SIR compartment model.

- Feedback-Loops $^{\S\mp}$ - because agents act continuously and their actions influence each other and themselves, feedback-loops are the norm in ABS.
- Heterogeneity $^{\mp}$ - 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. It is a very well studied and understood compartment model from epidemiology [3] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles [4] 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 interaction between infected persons does not lead to re-infection, thus these interactions are ignored 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 System Dynamics (SD) approach [5] which models a system through differential equations. For the SIR model we get the following equations:

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -infectionRate \tag{1}$$

$$\frac{\mathrm{d}I}{\mathrm{d}t} = infectionRate - recoveryRate \tag{2}$$

$$\frac{\mathrm{d}R}{\mathrm{d}t} = recoveryRate \tag{3}$$

$$infectionRate = \frac{I\beta S\gamma}{N} \tag{4}$$

$$recoveryRate = \frac{I}{\delta} \tag{5}$$

Solving these equations is then done 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*. 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.

$$S(t) = N + \int_0^t -infectionRate\,\mathrm{d}t \tag{6}$$

$$I(t) = 1 + \int_0^t infectionRate - recoveryRate\,\mathrm{d}t \tag{7}$$

$$R(t) = \int_0^t recoveryRate\,\mathrm{d}t \tag{8}$$

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.

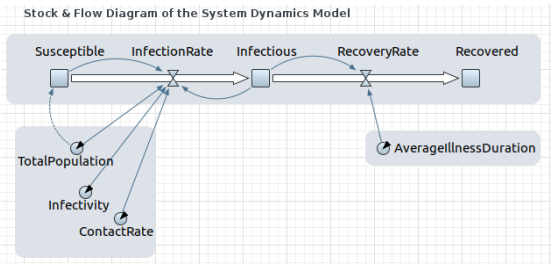

Fig. 2: A visual representation of the SD stocks and flows of the SIR compartment model. Picture taken using AnyLogic Personal Learning Edition 8.1.0.

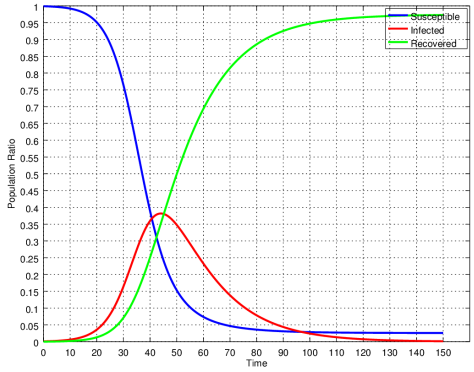

Fig. 3: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

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. Such questions were asked before and modelling the SIR model using an agent-based approach is indeed possible. It is important to note that SD can be seen as operating on averages thus treating 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 discretize the population and model each person in the population as an individual agent. The transition between the states are no longer happening according to continuous differential equations but due to discrete events caused both by interactions amongst the agents and time-outs.

TODO: this is already a too technical explanation which fixes the implementation details already on messaging / data-flow - this is too early and in deriving our approach we will implement 4 different approaches (feedback of all agent-states, data-flow, environment, transactions) TODO: the main point is that we are implementing a state-chart with the transitions are the main thing to consider

- 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 we need to sample from an exponential CDF because the rate is proportional to the size of the population as [6] pointed out.
- 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 which allows agents to interact is perfectly suited to do this.
  - *Susceptibles*: These agents make contact with other random agents (excluding themselves) with a "Susceptible" message. They can be seen to be the drivers of the dynamics.
  - *Infected*: These agents only reply to incoming "Susceptible" messages with an "Infected" message to the sender. Note that they themselves do *not* make contact pro-actively but only react to incoming one.
  - *Recovered*: These agents do not need to send messages because contacting it or being contacted by it has no influence on the state.

- Transition of susceptible to infected state - a susceptible agent needs to have made contact with an infected agent which happens when it receives an "Infected" message. If this happens an infection occurs with a probability of γ. The infection can be calculated by drawing p from a uniform random-distribution between 0 and 1 - infection occurs in case of $\gamma >= p$. Note that this needs to be done for *every* received "Infected" message.
- Transition of infected to recovered - a person recovers *on average* after δ time unites. This is implemented by drawing the duration from an exponential distribution [6] with $\lambda = \frac{1}{\delta}$ and making the transition after this duration.

For a more in-depth introduction of how to approximate an SD model by ABS see [7] who discusses a general approach and how to compare dynamics and [6] which explain the need to draw the illness-duration from an exponential-distribution. For comparing the dynamics of the SD and ABS approach to real-world epidemics see [8].

IV. Deriving a functional approach

In this section we will derive a functional approach for implementing an agent-based simulation of the SIR model. We will start out with a very naive approach and show its limitations which can be overcome by bringing in FRP. Then in three steps we will add more concepts and generalisations, ending up at the final approach which utilises monadic stream functions (MSF) [9], a generalisation of FRP. Although we presented a high-level agent-based approach to the SIR model in the previous section, which focused only on the states and the transitions, we haven't talked about technical implementation details on how to actually implement such a state-machine. In these steps we will ultimately present four different approaches on how to implement these states and transitions. Although all result *on average* in the same dynamics, not all of them are equally expressive and testable.

Step I: Naive beginnings

In our first step we start with modelling the states of the agents for which we simply use an Algebraic Data Type (ADT):

```
data SIRState = Susceptible | Infected | Recovered
```

Also agents are ill for some duration meaning we need to keep track when a potentially infected agent recovers. Also as previously mentioned, a simulation is stepped in discrete or continuous time-steps thus we introduce a notion of *time* and Δt by defining:

```
type Time      = Double
type TimeDelta = Double
```

Now we can represent every agent simply as a tuple of its SIR-state and its potential recovery time. We hold all our agents simply in a list and define helper functions:

```
type SIRAgent = (SIRState, Time)
type Agents   = [SIRAgent]

is :: SIRState -> SIRAgent -> Bool
is s (s',_) = s == s'

susceptible :: SIRAgent
susceptible = (Susceptible, 0)

infected :: Time -> SIRAgent
infected t = (Infected, t)

recovered :: SIRAgent
recovered = (Recovered, 0)
```

Next we need to think about how to actually step our simulation. For this we define a function which simply steps our simulation with a fixed Δt until a given time t where in each step the agents are processed and the output is fed back into the next step. TODO: need a much better explanation and maybe split up into more steps? As already mentioned in previous sections, the agent-based implementation of the SIR model is inherently stochastic which means we need access to a random-number generator. We decided to use the Rand Monad at this point as threading a generator through the simulation and the agents is very cumbersome. Thus our simulation stepping runs in the Rand Monad:

```
runSimulation :: RandomGen g
              => Time
              -> TimeDelta
              -> Agents
              -> Rand g [Agents]
runSimulation tEnd dt as = runSimulationAux 0 dt as []
  where
    runSimulationAux :: RandomGen g
                     => Time
                     -> TimeDelta
                     -> Agents
                     -> [Agents]
```

```
                    -> Rand g [Agents]
    runSimulationAux t dt as acc
      | t >= tEnd = return $ reverse (as : acc)
      | otherwise = do
        as' <- stepSimulation dt as
        runSimulationAux (t + dt) dt as' (as : acc)

stepSimulation :: RandomGen g => TimeDelta -> Agents -> Rand g Agents
stepSimulation dt as = mapM (processAgent dt as) as
```

Now we can implement the behaviour of an individual agent. First we need to distinguish between the agents SIR-states:

```
processAgent :: RandomGen g
             => TimeDelta
             -> Agents
             -> SIRAgent
             -> Rand g SIRAgent
processAgent _  as    (Susceptible, _) = susceptibleAgent as
processAgent dt _   a@(Infected   , _) = return $ infectedAgent dt a
processAgent _  _   a@(Recovered  , _) = return a
```

An agent gets fed all the agents states so it can draw random contacts. Note that this includes also the agent itself thus we would need to omit the agent itself to prevent making contact with itself. We decided against that as it complicates the solution and for larger numbers of agent population the probability for an agent to make contact with itself is so small that it can be neglected.

From our implementation it becomes apparent that only the behaviour of a susceptible agent involves randomness and that a recovered agent is simply a sink: it does nothing - its state stays constant.

Lets look how we can implement the behaviour of a susceptible agent. It simply makes contact on average with a number of other agents and gets infected with a given probability if an agent it has contact with is infected. When the agent gets infected it calculates also its time of recovery by drawing a random number from the exponential distribution meaning it is ill on average for illnessDuration.

```
susceptibleAgent :: RandomGen g => Agents -> Rand g SIRAgent
susceptibleAgent as = do
    rc <- randomExpM (1 / contactRate)
    cs <- doTimes (floor rc) (makeContact as)
    if elem True cs
      then infect
      else return susceptible

  where
    makeContact :: RandomGen g => Agents -> Rand g Bool
    makeContact as = do
      randContact <- randomElem as
      if (is Infected randContact)
        then randomBoolM infectivity
        else return False

    infect :: RandomGen g => Rand g SIRAgent
    infect = do
      randIllDur  <- randomExpM (1 / illnessDuration)
      return $ infected randIllDur
```

The infected agent is trivial. It simply recovers after the given illness duration which is implemented as follows:

```
infectedAgent :: TimeDelta -> SIRAgent -> SIRAgent
infectedAgent dt (_, t)
    | t' <= 0   = recovered
    | otherwise = infected t'
  where
    t' = t - dt
```

1) Results: We run the simulation for $t = 150$ time-units with a fixed $\Delta t = 1.0$. With increasing number of agents the dynamics approach the one of the SD simulation. TODO: add pictures

Reflecting on our first naive approach we can conclude that it already introduced the most fundamental concepts of ABS

- Time - the simulation occurs over virtual time which is modelled explicitly divided into *fixed* Δt where at each the agents are executed.
- Agents - we implement each agent as an individual behaviour which depends on the agents state.
- Feedback - the output state of the agent in the current time-step t is the input state for the next time-step $t + 1$.
- Environment - as environment we implicitly assume a fully-connected network where every agent 'knows' every other agents, including itself and thus can make contact with every other agent (including itself).
- Stochasticity - its an inherently stochastic simulation, which is indicated by the Rand Monadic type.

- Deterministic - repeated runs with the same initial random-number generator result in same dynamics. This may not come as a suprise but in Haskell we can guarantee that property statically already at compile time because our simulation runs in the Rand monad and NOT in the IO Monad. This guarantees that no external, uncontrollable sources of randomness can interfere with the simulation.
- Dynamics - it works as expected: with increasing number of agents our solution approaches the SD dynamics

Nonentheless our approach has also weaknesses and dangers:

- Δt is passed explicitly as argument to the agent and needs to be dealt with explicitly. It seems to be not very elegant and a potential source of errors - can we do better and find a more elegant solution?
- The states of all agents of the current step are fed back into every agent in the next step so that an agent can pick its contacts. Although agents cannot change the states, this reveals too much information e.g. the illness durtion is of no interest to the other agents. Although we could just feed in the SIRStates without the illness duration, the problem is more of conceptual nature: it should be the agent which decides to whom it revealse which information.
- The way our agents are represented is not very elegant: the state of the agent is explictly encoded in an ADT and when processing the agent the function needs always first distinguish between the states. Can we express it in a more implicit, functional way?

A. Step II: Adding FRP

- time is implicit and cannot be messed with - agent can switch their behaviour

B. Step III: Adding data-flow

- getting rid of feeding all agent-states back into every agent, making data-flows explicit between agents, which is necessary because connections between agents are not fixed at compile time - with occasionally we can achieve extremely fine grained stochastics as opposed to draw random number of events we create only a single event or not, this allows for a much smoother curve and is a real advantage: we are treating it as a continuous system

C. Step IV: Generalising to Monadic Stream Functions

- this allows us to add dynamic environments and agent transactions - we need deterministic behaviour under all curcumstances, thus we cannot use IO or STM. for globally mutable state we use StateT - also putting agentout into StateT monad composes now much better

D. Step V: Adding an environment

E. Step VI: Adding agent transactions

ACKNOWLEDGMENTS

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

REFERENCES

[1] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, 2009.
[2] J. M. Epstein, *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton University Press, Jan. 2012, google-Books-ID: 6jPiuMbKKJ4C.
[3] W. O. Kermack and A. G. McKendrick, "A Contribution to the Mathematical Theory of Epidemics," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 115, no. 772, pp. 700–721, Aug. 1927. [Online]. Available: http://rspa.royalsocietypublishing.org/content/115/772/700
[4] R. H. Enns, *It's a Nonlinear World*, 1st ed. Springer Publishing Company, Incorporated, 2010.
[5] D. E. Porter, "Industrial Dynamics. Jay Forrester. M.I.T. Press, Cambridge, Mass.; Wiley, New York, 1961. xv + 464 pp. Illus. $18," *Science*, vol. 135, no. 3502, pp. 426–427, Feb. 1962. [Online]. Available: http://science.sciencemag.org/content/135/3502/426.2
[6] A. Borshchev and A. Filippov, "From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools," Oxford, Jul. 2004.
[7] 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
[8] A. Ahmed, J. Greensmith, and U. Aickelin, "Variance in System Dynamics and Agent Based Modelling Using the SIR Model of Infectious Disease," *arXiv:1307.2001 [cs]*, Jul. 2013, arXiv: 1307.2001. [Online]. Available: http://arxiv.org/abs/1307.2001
[9] 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