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. 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 which is based on our *FrABS* library. 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 library. We then discuss the benefits and drawbacks of our approach and compare it to existing methodology. The contribution is a novel approach to implementing ABS with more emphasis on specification and possibilities to reason about the correctness of the simulation.

Fig. 1: Transitions in the SIR compartment model.

II. Background

A. 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* $^{\mp}$, 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.
- 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.

B. Functional Reactive Programming

TODO: short introduction, following all the other papers which is basically useless because it only scratches the surface... For a good introduction into FRP using Yampa we refer to the papers of [11], [12] and [13].

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 System Dynamics (SD) approach which models a system through differential equations. For the SIR model we get the following equations:

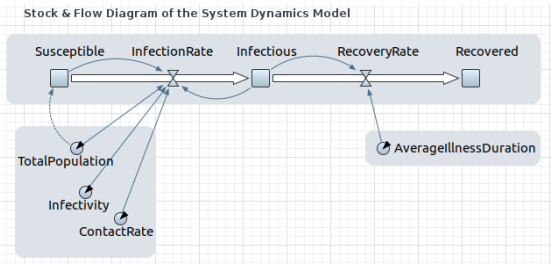


Fig. 2: A visual representation of the SD stocks and flows of the SIR compartment model. Picture taken using AnyLogic Personal Learning Edition 8.1.0.

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -infectionRate$$
$$\frac{\mathrm{d}I}{\mathrm{d}t} = infectionRate - recoveryRate$$
$$\frac{\mathrm{d}R}{\mathrm{d}t} = recoveryRate$$

$$infectionRate = \frac{I\beta S\gamma}{N}$$
$$recoveryRate = \frac{I}{\delta}$$

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* [1].

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

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.

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 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.

- 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 [14] 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-mechanism which allows agents to interact is perfectly suited to do this.

[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.

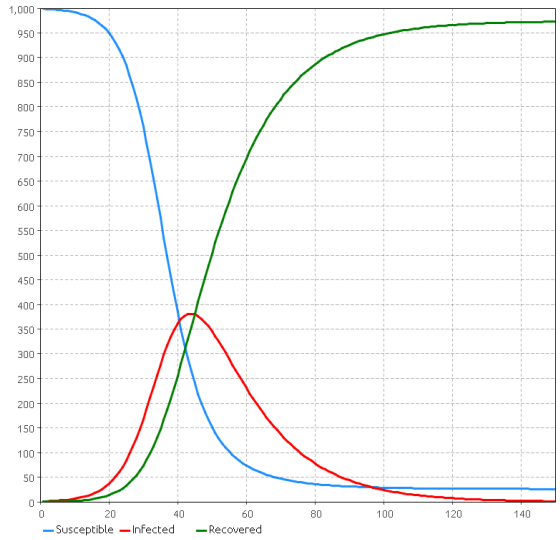


Fig. 3: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1000, contact rate $\beta = 1/5$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run of 150 time-steps. Dynamics generated with AnyLogic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

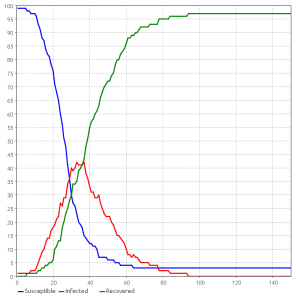
  - *Susceptible* agent: there is actually no need for a susceptible agent to contact others as making contact with a susceptible agent has no influence on the state of the contacted agent.
  - *Infected* agent: sends an "Infected" message to other agents as described above.
  - *Recovered* agent: does 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 [14] with $\lambda = \frac{1}{\delta}$ and making the transition after this duration.

We will derive the implementation of this approach in the following sections and finally present it in our FrABS library. As will be seen the library allows us express this behaviour very explicitly and is 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.

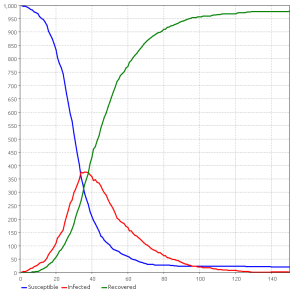
TODO: replace these pictures with ones generated by FrABS

TODO: reproducing about the same dynamics of the SD-solution (1.0 dt) - super-sampling: contact-rate ss high, illness time-out low - agent number: 1000 vs. 10.000 agents - Susceptibles making contact and infected response VS. only Infected make contact - update-strat: Sequential vs. Parallel - making contact: susceptible only vs. susceptible AND infected - do conversations make a difference?

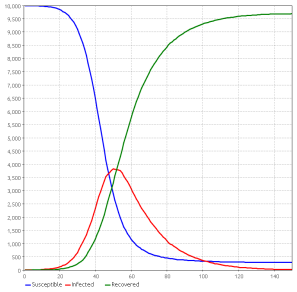
As previously mentioned the agent-based approach is a discrete one which means that with increasing number of agents, the discrete dynamics approximate the continuous dynamics of the SD simulation. Still the dynamics of 10,000 Agents do not match the dynamics of the SD simulation to a satisfactory level 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 matches the



(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 ($\beta = 1/5$, $\gamma = 0.05$, $\delta = 15$ with initially 1 infected agent) except population size. All simulations run for 150 time-steps. Dynamics generated with AnyLogic Personal Learning Edition 8.1.0. TODO: replace by my own pictures

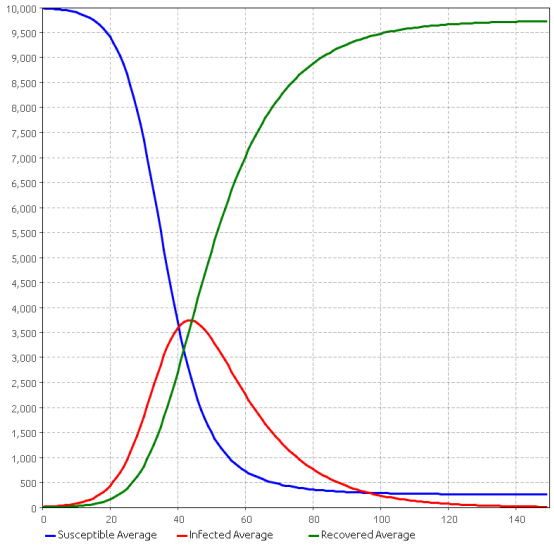


Fig. 5: Dynamics of the SIR compartment model using the agent-based approach with same parameters as in SD ($\beta = 1/5$, $\gamma = 0.05$, $\delta = 15$) but with a population of 10,000 where 10 are initially infected and the dynamics averaged over 1000 replications. Dynamics generated with AnyLogic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

SD dynamics to a very satisfactory level [2].

TODO: replace by my own pictures

For a more in-depth introduction of how to approximate an SD model by ABS see [15] who discusses a general approach and how to compare dynamics and [14] which explain the need to draw the illness-duration from an exponential-distribution.

IV. Functional Reactive ABS

TODO: cross reference to the code in the appendix by giving line-numbers e.g. the full implementation of a susceptible agent can be seen in

The challenges one faces when implementing an ABS plain, without support from a library are manifold. In the paper (TODO: cite my own paper on update-strategies) the authors discuss already the fundamental things to consider in a programming language agnostic way. Generally one faces the following challenges:

[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 in which 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.

- Agent Representation - how do we represent an agent? The ABS community implements agents as objects (as in Java, Python or C++) as they claim that the mapping of an agent on an object is natural. The question is how to represent an agent in Haskell?
- Agent-Agent Interaction - how can agents interact with other agents? In object-orientation we have method-calls which allows to call other objects and mutate their state. Also this is not available in Haskell, so how do we solve this problem without resorting to the IO monad?
- Environment representation - how can we represent an environment? Also an environment must have the ability to update itself e.g. regrow some resources.
- Agent-Environment interaction - how can agents interact (read / write) with the environment they are situated in?
- Agent Updating - how is the set of agents organised, how are they updated and how is it managed (deleting, adding during simulation)? In object-oriented implementations due to side-effects and mutable data in-order updates are easily done but this is not available in Haskell without resorting to the IO monad.

In the next subsections we will discuss each point by deriving a functional reactive implementation of the agent-based SIR model. For us it is absolutely paramount that the simulation should be pure and never run in the IO Monad (except of course the surrounding Yampa loop which allows rendering and output). The complete source-code can be seen in Appendix TODO: ref.

A. Agent Representation

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 the agent is situated in and can interact with
- **b** - the behaviour of the agent

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.

Each agent may have a generic state which could be any data type or compound data. A SIR agent's state can be represented using the an AGDT as follows:

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

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

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

AgentIn provides the necessary data to the agent-behaviour: its id, incoming messages, the current state s and a random-number generator.

AgentOut allows the agent to communicate changes: kill itself, create new agents, sending messages, an updated state s and a changed random-number generator.

The behaviour also gets the environment passed in, which the agent can read and also write by changing it and returning it along side the AgentOut. It is important to note, that the environment is completely generic and we do not induce any type-bounds on it.

Obviously AgentIn is read-only [3] whereas AgentOut is both read- and write-able. The first thing an agent-behaviour does is creating the default AgentOut from the existing AgentIn using the function.

```
agentOutFromIn :: AgentIn s m e -> AgentOut s m e
```

This will copy the relevant fields over to AgentOut on which one now primarily acts. The read-only and read/write character of both types is also reflected in the EDSL where most of the functions implemented also work that way: they may read the AgentIn and read/write an AgentOut.

Relevant functions for working on the agent-definition are:

```
kill :: AgentOut s m e -> AgentOut s m e
isDead :: AgentOut s m e -> Bool
onStart :: (AgentOut s m e -> AgentOut s m e) -> AgentIn s m e -> AgentOut s m e -> AgentOut s m e

setDomainState :: s -> AgentOut s m e -> AgentOut s m e
updateDomainState :: (s -> s) -> AgentOut s m e ->  AgentOut s m e

createAgent :: AgentDef s m e -> AgentOut s m e -> AgentOut s m e
```

[3] Of course one could change it, but the changes are never propagated out of the function thus ultimately making them never happen due to lazy evaluation.

The function *kill* marks an agent for removal after the current iteration. The function *isDead* checks if the agent is marked for removal. The function *onStart* allows to change the AgentOut in the case of the start-event which happens on the very first time the agent runs. The function *setDomainState* allows to change the state of the agent by overriding it and *updateDomainState* allows to change it by keeping parts of it. The function *createAgent* allows to add an agent-definition [4] to the AgentOut which results in creating a new agent from the given definition which will be active in the next iteration of the simulation.

Having these functions we build some reactive primitives into our EDSL meaning that they return signal-functions themselves. We start with the following functions:

```
doOnce :: (AgentOut s m e -> AgentOut s m e) -> SF (AgentOut s m e) (AgentOut s m e)
doNothing :: AgentBehaviour s m e

setDomainStateR :: s -> AgentBehaviour s m e
updateDomainStateR :: (s -> s) -> AgentBehaviour s m e
```

The *doOnce* function may seem strange at first but allows conveniently make actions (which are changinge the agent-out) only once e.g. when making the transition from Susceptible to Infected changing the state to Infected just once. A more striking example would be to send a message just once after a transition. The function *doNothing* provides a convenient way of an agent-sink which is basically an agent which does literally nothing - the resulting agent-behaviour just transforms the agent-in to agent-out using the previously mentioned function *agentOutFromIn*.

Often we want some more reactive behaviour e.g. making a transition from one behaviour to another on a given event. For this we provide the following:

```
type EventSource s m e = SF (AgentIn s m e, AgentOut s m e) (AgentOut s m e, Event ())
transitionOnEvent :: EventSource s m e -> AgentBehaviour s m e -> AgentBehaviour s m e -> AgentBehaviour s m e
```

The function *transitionOnEvent* takes an event-source which creates the event, an agent-behaviour which is run until the event hits and an agent-behaviour which is run at the event and after. The event-source is a signal-function itself to allow maximum of flexibility and gets both agent-in and agent-out and returns a (potentially changed) agent-out and the event upon to switch. The workings of this functionality can be seen for implementing the susceptible agent where we use a *transitionOnEvent* and a specific event-source which generates an event when the susceptible agent got infected.

Sometimes we need our transition event to rely on time-semantics e.g. in SIR where an infected agent recovers *on average* after δ time-units. For this we provide the following function

```
transitionAfterExp :: RandomGen g => g
                                  -> Double
                                  -> AgentBehaviour s m e
                                  -> AgentBehaviour s m e
                                  -> AgentBehaviour s m e
```

It takes a random-number generator, the *average* time-out, the behaviour to run before the time-out and the behaviour to run after the time-out where the function will return then the according behaviour. For implementing this behaviour we initially used Yampas *after* function which generates an event after given time-units but this would not result in the correct dynamics as we rather need to create a random-distribution of time-outs than a deterministic time-out which occurs always after the same time. For this we implemented our own function, called *afterExp*, which now takes a random-number generator a time-out and some value of type b and creates a signal-function which ignores its input and creates an event *on average* after DTime.

```
afterExp :: RandomGen g => g -> DTime -> b -> SF a (Event b)
```

B. Agent-Agent Interaction

Agent-agent interaction is the means of an agent to directly address another agent and vice versa. Inspired by the actor model we implement a *messaging* with share nothing semantics. In this case the agent send messages which will arrive in the next step of the simulation at the receiver thus being kind of asynchronous - a round-trip would always take at least two steps, independent of the sampling time. Depending on the semantics of the model we sometimes need synchronous interactions e.g. when only one agent can change the environment or decisions need to be made within one step - this wouldn't be possible with the asynchronous messaging. For this we introduced the concept of *conversations* which allow two agents to interact with each other for an arbitrary number of requests and replies without the simulation being advanced - time is halted and only the two agents are active until they finish their conversation.

1) Messaging: Each Agent can send a message to an other agent through AgentOut-Signal where incoming messages are queued in the AgentIn-Signal and can be processed when the agent is active the next time. The agent is free to ignore the messages and if it does not process them they will simply be lost. This is in fundamental contrast to the actor model where messages stay in the message-box of the receiving actor until the actor has processed them. We chose a different approach as time has a different meaning in ABS than in a system of actors where there is basically no global notion of time. 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

[4] AgentDef simply contains the initial state, behaviour and id of the agent (amongst others).

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. We need a set of messages *m* the agents understand. In the case of the SIR model we simply use the following:

```
data SIRMsg = Contact SIRState
```

In addition we provide the following functions in our EDSL to support messaging.

```
type AgentMessage m = (AgentId, m)

sendMessage :: AgentMessage m -> AgentOut s m e -> AgentOut s m e
sendMessageTo :: AgentId -> m -> AgentOut s m e -> AgentOut s m e
sendMessages :: [AgentMessage m] -> AgentOut s m e ->  AgentOut s m e
broadcastMessage :: m -> [AgentId] -> AgentOut s m e -> AgentOut s m e

hasMessage :: (Eq m) => m -> AgentIn s m e -> Bool
onMessage :: (AgentMessage m -> acc -> acc) -> AgentIn s m e -> acc -> acc
onMessageFrom :: AgentId -> (AgentMessage m -> acc -> acc) -> AgentIn s m e -> acc -> acc
onMessageType :: (Eq m) => m -> (AgentMessage m -> acc -> acc) -> AgentIn s m e -> acc -> acc
```

Most of the functions are pretty self-explanatory, we will shortly explain the *onMessage**. The function *onMessage* provides a way to react to incoming messages by using a callback function which manipulates an accumulator, thus resembling the workings of fold. The functions *onMessageFrom* and *onMessageType* provide the same functionality but filter the messages accordingly. We can now write a function which allows an agent to reply to an incoming Contact message with another contact message of a given SIRstate. TODO: refer to the appendix and give line-numbers instead of inserting it here.

Sometimes we also need discrete semantics like changing the behaviour of an agent on reception of a specific message. For this we provide the function *transitionOnMessage* which works the same way as *transitionOnEvent* but now on a message instead.

```
transitionOnMessage :: (Eq m) => m -> AgentBehaviour s m e -> AgentBehaviour s m e -> AgentBehaviour s m e
```

Of course messaging sometimes may have specific time-semantics as in our SIR model. There susceptible agents make contact with n other agents on average *per time unit*. To implement this we randomly need to generate messages with a given frequency within some time-interval by drawing from the exponential random-distribution. This is already supported by Yampa using *occasionally* and we have built on it a the following:

```
type MessageSource s m e = e -> AgentOut s m e -> (AgentOut s m e, AgentMessage m)

sendMessageOccasionallySrc :: RandomGen g => g
                              -> Double
                              -> MessageSource s m e
                              -> SF (AgentOut s m e, e) (AgentOut s m e)

constMsgReceiverSource :: m -> AgentId -> MessageSource s m e
randomNeighbourNodeMsgSource :: m -> MessageSource s m (Network l)
randomNeighbourCellMsgSource :: (s -> Discrete2dCoord) -> m -> Bool -> MessageSource s m (Discrete2d AgentId)
randomAgentIdMsgSource :: m -> Bool -> MessageSource s m [AgentId]
```

The function *sendMessageOccasionallySrc* takes a random-number generator, the frequency of messages to generate *on average per time-unit* a message-source and returns a signal-function which takes a tuple of an agent-out and environment and returns an agent-out. This signal-function which performs the actual generating of the messages needs to be fed in the tuple but only returns the changed agent-out but not the environment - this guarantees statically at compile-time that the environment cannot be changed in this process. This is also directly reflected in the type of MessageSource which takes an environment and agent-out and returns a tuple with a changed agent-out and a message. We provide pre-defined messages-sources like *constMsgReceiverSource* which always generates the same message, *randomNeighbourNodeMsgSource* which picks a random neighbour from a network-environment (see below), *randomNeighbourCellMsgSource* which picks a random neighbour from a discrete 2d-grid environment (see below) and *randomAgentIdMsgSource* which randomly picks an element from an environment which is a list of AgentIds (omitting the sender True/False). As can be seen in TODO: refer to appendix the susceptible agent makes use of this function.

2) 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 resource-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.

```
type AgentConversationReply s m e = Maybe (m, AgentIn s m e, e)

type AgentConversationReceiver s m e = (AgentIn s m e
                                        -> e
                                        -> AgentMessage m
                                        -> AgentConversationReply s m e)

type AgentConversationSender s m e = (AgentOut s m e
                                      -> e
                                      -> Maybe (AgentMessage m)
                                      -> (AgentOut s m e, e))

conversation :: AgentMessage m
                -> AgentConversationSender s m e
                -> AgentOut s m e
                -> AgentOut s m e

conversationEnd :: AgentOut s m e -> AgentOut s m e
```

The agent-conversations sender is the initiator of the conversation which can only be an agent which is just run and has started a conversation with a call to the function *conversation*. After the agent has run, the simulation-system will detect the request for a conversation and start processing it by looking up the receiver and calling the functions with passing the values back and forth. While a conversation is active, time does not advance and other agents do not act. Note that due to its nature, conversations are only available when using the *sequential* update-strategy (see below).

C. Environment representation

Agents are situated within an environment which itself is *not* an agent - although it can have its own behaviour e.g. regrowing resources, it cannot send or receive messages from agents. Thus we treat the environment completely generic by allowing any given type captured in the type-variable *e*. Environment behaviour is optional but if required, implemented using a signal-function which simply maps *e* to *e*:

```
type EnvironmentBehaviour e = SF e e
```

By allowing a signal-function as the environment behaviour gives us the opportunity to implement reactive behaviour and time-semantics in environments as well. Again it is essential to note that throughout the whole simulation implementation we never put any bounds on the environment nor make assumptions about its type (except for providing utility-functions which allow agents to operate on existing environment implementations).

Although environments can be anything (even be of unit-type, if no environment is required at all), there exist a few standard environments in ABS which are provided in all ABS packages. We provide implementations for them and discuss them below. Note that we don't provide the APIs of the environments here as it is out-of-scope of this paper.

1) Network: A network environment gives agents access to a network, represented by a graph where the nodes are agent-ids and the edges represent neighbourhood information. We implemented fully-connected, Erdos-Renyi and Barbasi-Albert networks. In our case the networks are undirected and the labels can be labelled, carrying arbitrary data or being unlabelled, having unit-type. Agents can then perform the usual graph algorithms on these networks.

2) Discrete 2D: A discrete 2d environment gives agents access to a 2d-grid with dimensions of $N \times M \in \mathbb{N}$ cells. The cells are of a generic type c and can thus be anything from AgentIds to resource-sites with single or multiple occupants. Such an environment has a defined neighbourhood of either Moore (8 neighbours) or Von-Neumann (4 neighbours). Agents can then query the environment for cells using neighbourhoods, radius or specific positions, change the cells and update them in the environment.

3) Continuous 2D: A continuous 2d environment gives agents access to a continuous 2d-space with dimensions of $N \times M \in \mathbb{R}$. This space can contain an arbitrary number of objects of the generic type o where each of them has a coordinate within this space. Agents can query for objects within a given radius, add, remove and update them. Also we provide functions to move objects either in a given or random direction.

D. Agent-Environment interaction

In ABS agents almost always are situated within an environment. We follow a subtle different approach and implement it in a way that agents have access to a generic environment of type e as discussed above instead of being situated within. It is important to note the subtle difference of agents *having access* to the environments instead of *being situated* within them. This allows to free us making assumptions within an environment how agents use these environments and also allows us to stack multiple environments e.g. agents moving on a discrete 2d-grid but relying on neighbourhood from a network. Our SIR implementation uses a list of all AgentIds as the environment which means that every agent knows all the existing agents of the simulation and can address them. We could have used a Network environment using a fully-connected graph but the memory-consumptions of the library FGL we are using for graphs are unacceptable in case of fully-connected networks of a larger numbers of agents (10,000). Each agent gets the environment passed in through the AgentIn and can change it by passing a changed version of the environment out through AgentOut.

E. Agent Updating

For agents to be pro-active, they need to be able to perceive time. Also agents must have the opportunity to react to incoming messages and manipulate the environment. The work of (TODO: cite our own paper on update-strategies) identified four possible ways of doing this where we only implemented the *sequential-* and *parallel-strategy* [5]. Implementing these 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. We implement both update-strategies basically by running all agents behaviour signal-functions every δt, so when running a simulation for a duration of t the number of steps is $\frac{t}{\delta t}$. It is important to realise that in our approach of a single behaviour function we merge both pro-activity, time-dependent behaviour and message-receiving behaviour. A different approach would be to have callbacks for messages in addition to the normal agent-behaviour but this would be quite cluttered and unelegant.

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).

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 the next agent i + 1 AgentIn in the current iteration. 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. As previously mentioned, conversations are *only* possible within this update-strategy because only in this strategy agents act after another which is a fundamental requirement for conversations to make sense and work correctly.

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. Each agent receives a copy of the environment upon which it can work and return a changed one. 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 the model implementer needs to provide a corresponding function.

```
type EnvironmentCollapsing e = [e] -> e
```

Of course not all models have environments which can be changed and in the SIR model we indeed use a list of AgentIds which won't change during execution (meaning, the agents only read it). Because of this environment-collapsing functions are optional and when none is provided the environments returned by the agents are ignored and always the initial one is provided. To make this more explicit we introduce a wrapper which wraps a signal-function which is the same as agent-behaviour but omits the environment from the out tuples. When this wrapper is used one can guarantee statically at compile-time that the environment cannot be changed by the agent-behaviour. We also provide a function which completely ignores the environment, which allows to reason already at compile time that no environment access will happen ever.

[5]The other two strategies being the *concurrent-* and *actor-strategy*, both requiring to run within the STM-Monad, which is not possible with Yampa. The work of Ivan Perez in [16] implemented a library called Dunai, which is the same as Yampa but capable of running in an arbitrary Monad. We leave this for further research.

```
type ReactiveBehaviourIgnoreEnv s m e = SF (AgentIn s m e) (AgentOut s m e)
type ReactiveBehaviourReadEnv s m e = SF (AgentIn s m e, e) (AgentOut s m e)

ignoreEnv :: ReactiveBehaviourIgnoreEnv s m e -> AgentBehaviour s m e
readEnv :: ReactiveBehaviourReadEnv s m e -> AgentBehaviour s m e
```

We use both functions in our SIR implementation TODO: refer to appendix.

F. Non-Reactive Features

Not all models have such explicit time-semantics as the SIR model. Such models just assume that the agents act in some order but don't rely on any time-outs, timed transitions or rates. These models are more of imperative nature and map therefore naturally to a monadic style of programming using the do-notation. Unfortunately we don't have this functionality available within a SF thus to support the programming of such imperative models, we implemented wrapper-functions which allow to provide both non-monadic and monadic functionality which runs within a wrapper-signal function.

1) Non-monadic (pure) wrapping:

```
agentPure :: (e -> Double -> AgentIn s m e -> AgentOut s m e -> (AgentOut s m e, e)) -> AgentBehaviour s m e
agentPureReadEnv :: (e -> Double -> AgentIn s m e -> AgentOut s m e -> AgentOut s m e) -> AgentBehaviour s m e
agentPureIgnoreEnv :: (Double -> AgentIn s m e -> AgentOut s m e -> AgentOut s m e) -> AgentBehaviour s m e
```

The function *agentPure* wraps a non-monadic (pure) function and wraps it in a SF. This pure function has the following arguments: the environment, the current local time of the agent, the AgentIn, a default AgentOut and must return the tuple of AgentOut and the environment.

The function *agentPureReadEnv* works the same as *agentPure* but omits the environment from the return-type thus ensuring statically at compile-time that an agent which implements its behaviour in this way can only read the environment but never change it.

The function *agentPureIgnoreEnv* is an even stricter version of *agentPureReadEnv* and omits environment from the agents behaviour altogether thus ensuring statically at compile-time that an agent which wraps its behaviour in this function will never access the environment.

2) Monadic wrapping: The monadic wrappers work basically the same way as the pure version, with the difference that they run within the state-monad with AgentOut being the state to pass around.

```
agentMonadic :: (e -> Double -> AgentIn s m e -> State (AgentOut s m e) e) -> AgentBehaviour s m e
agentMonadicReadEnv :: (e -> Double -> AgentIn s m e -> State (AgentOut s m e) ()) -> AgentBehaviour s m e
agentMonadicIgnoreEnv :: (Double -> AgentIn s m e -> State (AgentOut s m e) ()) -> AgentBehaviour s m e
```

We also provide monadic versions of the pure primitives of our EDSL which work on AgentOut, discussed above. They run in the state-monad where the state is the agent-out as this is the data which is manipulated step-by-step for the final output.

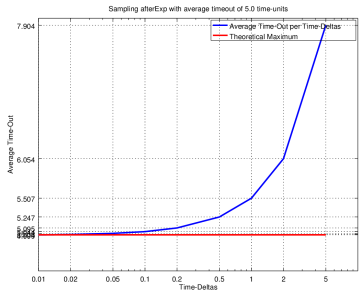
```
agentIdM :: State (AgentOut s m e) AgentId

sendMessageM :: AgentMessage m -> State (AgentOut s m e) ()
sendMessageToM :: AgentId -> m -> State (AgentOut s m e) ()
onMessageM :: (Monad mon) => (acc -> AgentMessage m -> mon acc) -> AgentIn s m e -> acc -> mon acc

createAgentM :: AgentDef s m e -> State (AgentOut s m e) ()

killM :: State (AgentOut s m e) ()
isDeadM :: State (AgentOut s m e) Bool

getDomainStateM :: State (AgentOut s m e) s
updateDomainStateM :: (s -> s) -> State (AgentOut s m e) ()
setDomainStateM :: s -> State (AgentOut s m e) ()
domainStateFieldM :: (s -> t) -> State (AgentOut s m e) t
```

For the environment-behaviour we provide a monadic wrapper as well.

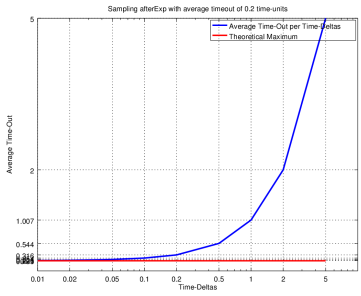
```
type EnvironmentMonadicBehaviour e = (Double -> State e ())
environmentMonadic :: EnvironmentMonadicBehaviour e -> EnvironmentBehaviour e
```

V. Sampling the System

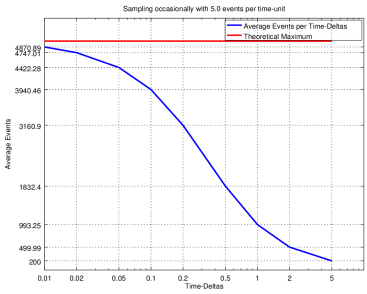
discuss the importance of sampling the system. When sampling the system, the correct time-delta must be selected which depends on the highest frequency which occurs in a time-reactive function in the whole system. For example in the FrSIR model we want infected agents to make on average contact with 5 other agents per time-unit, which means with a frequency of $frac15$. This functionality is built on Yampas function occasionally which behaviour we investigated under differing time-deltas with the above frequency. In this investigation we simply sampled occasionally with different time-deltas for a duration of 1000 time-units and the event-frequency of $frac15$. The results can be seen in Figure 6c and are quite striking. The plot clearly shows that occasionally needs quite high sampling frequency even for comparatively low event-frequency - this becomes of course worse for higher event-frequencies.



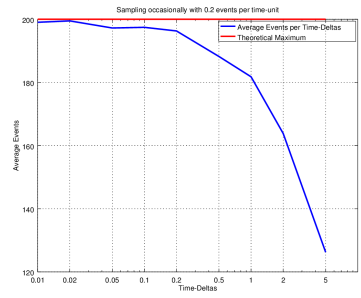
(a) Sampling *afterExp* with an average time-out of 5.



(b) Sampling *afterExp* with an average time-out of 0.2.



(c) Sampling *occasional* with a frequency of $\frac{1}{5}$ (average of 5 events per time-unit). The theoretical average is 5000 events within this time-frame.



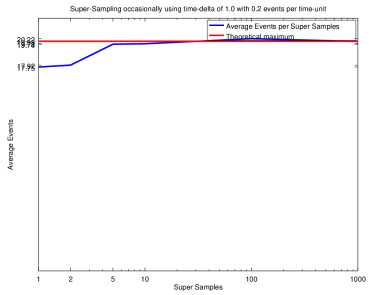
(d) Sampling *occasional* with a frequency of 5 (average of 0.2 events per time-unit). The theoretical average is 200 events within this time-frame.

Fig. 6: Sampling the *afterExp* and *occasional* functions to visualize the influence of sampling frequencies on the occurrence of the respective events. Time-deltas are [5, 2, 1, $\frac{1}{2}$, $\frac{1}{5}$, $\frac{1}{10}$, $\frac{1}{20}$, $\frac{1}{50}$, $\frac{1}{100}$]. The experiments for *afterExp* used 10,000 replications. The experiments for *occasional* ran for 1000 time-units with 100 replications.

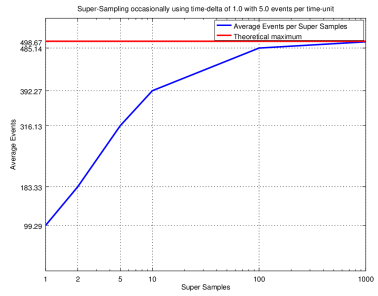
The other time-reactive function which occurs in the FrSIR model is the timed transition from infected to recovered which occurs on average with an exponential random-distribution after 15 time-units. This functionality is built on a custom implementation of Yampas after which creates an event after a time-out of the passed in time-duration drawn from an exponential random-distribution. Clearly this function has different semantics as although it also continuously emit events over time - NoEvent before the time was hit, and Event x after the time hit - the relevant point is that it switches to Event at some discrete point in time. This is implemented as simply adding up the time-deltas until the accumulator is GE than the previously drawn exponential time-out. We also investigated the behaviour of this function under varying time-deltas using a time-out of 15 (drawn from an exponential distribution within the function). Our approach was to sample the afterExp until an event occurs (this is one of the occasions where lazy evaluation really shines as one simply repeats the time-delta stream forever but then searches for the first occurrence of an event, which MUST occur at some point due to mathematical exponential distribution and our parameters to it, so it will always terminate) and then see when it has occurred. We run this with 10,000 replications with different random-number seeds and average the resulting times. The results can be seen in Figure 6a. The result is striking in another way: this function seems to be pretty invariant to the time-deltas, for obvious reasons: we are basically just interested in the ”after”-condition of the whole semantics whereas in occasionally we are interested in the ”repeatedly”-conditions. If we under the afterExp then we can be off by one time-delta. If we under sample occasionally we keep loosing events, the close time-delta and event-frequency are, the more we lose. Of course afterExp can also be used for very short time-outs e.g. $frac15$. We have investigated the behaviour of this function for various time-deltas as well as seen in Figure **??**. Here the result is much more striking and shows that afterExp is vulnerable to small time-outs as well as occasionally. To show that occasionally is not vulnerable to very low frequencies of e.g. one event every 5 time-steps we plotted the behaviour of this under varying time-steps in Figure 6d. The result shows that for low frequencies occasionally works fine with larger time-deltas

A. *Super-Sampling*

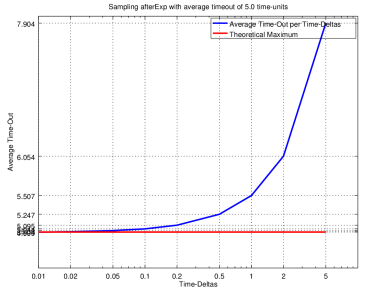
of course performance is a big issue and it decreases as time-deltas get smaller and smaller. if we could perform subsampling just for the given high-frequency function with the remaining system running in lower frequency then we could achieve substantial performance-increase. TODO: talk a little bit more about the costs of too small time-deltas: need to go through all agents all the time



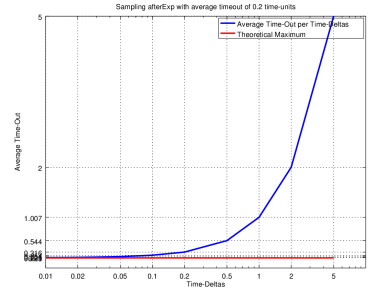
(a) Super-Sampling the *occasional* function with event-frequency of 5 (average of 0.2 events per time-unit). The theoretical average is 20 event within this time-frame.



(b) Super-Sampling the *occasional* function with event-frequency of $\frac{1}{5}$ (average of 5 events per time-unit). The theoretical average is 500 event within this time-frame.



(c) Super-Sampling the *afterExp* function with average time-out of 5.



(d) Super-Sampling the *afterExp* function with average time-out of 0.2.

Fig. 7: Super-Sampling the *afterExp* and *occasional* functions to visualize the influence of increasing number of super-samples on the average occurrence of the respective events. The time-delta is fixed to 1.0 in both cases with super-samples of [1, 2, 5, 10, 100, 1000]. The experiments for *afterExp* used 10,000 replications. The experiments for *occasional* ran for 100 time-units with 100 replications.

formula of calculating-steps = steps per time-unit * time-to-run-the-simulation

The problem is that *embed* does not really help because when running a sf with it, the signal-functions time does not advance. Thus we implemented a new signal-function which allows us to super-sample another signal-function.

```
superSampling :: Int -> SF a b -> SF a [b]
```

It takes the number of super-samples *n* and the signal-function *sf* to sample and returns a new signal-function which performs the super-sampling. It does this by evaluating sf for n times with dt of $\frac{dt}{n}$ and the same input argument a for all n evaluations. At time 0 no super-sampling is done and just a single output of sf is calculated. A list of b is returned with length of n containing the result of the n evaluations of sf. If 0 or less super samples are requested exactly one is calculated.

We ran tests super-sampling both *occasionally* (Figure 7a, Figure 7b) and *afterExp* (Figure). They work the same way as above except that now the time-delta is fixed to 1.0 but using increasing numbers of super-samples. The results are as expected: as the number of super-samples increase, so increases the accuracy.

At first this might not seem to be a real win as we still need to calculate a big number of samples every time. The big win comes though when these super-sampled signal-functions are embedded in a larger system which could run on a comparatively low frequency of e.g. 1 dt. So we are then increasing the sampling-frequency just where we need it and keep the frequency low where it is not required.

1) Super Sampling in SIR: We are using super-sampling in our SIR implementation to increase performance. We do this by keeping the time-delta at 1.0 and super-sampling the relevant functions with time-semantics which are *transitionAfterExp* and *sendMessageOccationallySrc*. For both we provide in our EDSL versions which support super-sampling:

```
sendMessageOccasionallySrcSS :: RandomGen g => g
                             -> Double
                             -> Int
                             -> MessageSource s m e
                             -> SF (AgentOut s m e, e) (AgentOut s m e)

transitionAfterExpSS :: RandomGen g =>
                    g
                    -> Double
```

```
    -> Int
    -> AgentBehaviour s m e
    -> AgentBehaviour s m e
    -> AgentBehaviour s m e
```

Both now take an additional parameter which determines the number of super-samples to be calculated. According to the above observations of the *occasionally* and *afterExp* functions which are the foundations of both of the functions we sample *sendMessageOccasionallySrcSS* with 20 super-samples and *transitionAfterExpSS* with 2. This will ensure that by using a time-delta of 1.0 we only calculate t steps when running a simulation for t time but that we sample our relevant functions with enough resolution to capture its frequencies. Optimally we should increase the number of super-samples for *sendMessageOccasionallySrcSS* to about 100 this will result in lower performance as *every* agent will perform this super-sampling, which will also result in a performance decrease. So in the end it is a struggle of performance vs. sufficiently close approximation.

VI. Discussion

A. Other Models

TODO: mention that we have also implemented other models, which also work without time-semantics (all agents make a move at discrete time-steps and do not really rely on some notion of time).

B. 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 given rate message transition: transition upon receiving a given message

C. 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.

D. 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

E. System Dynamics

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

F. 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

G. Advantages

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

H. Disadvantages

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 (this has been proved by example!)

VII. 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 [17], [18], [19]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [20]. It comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. [19] 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 [21]. 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 [6]. In games these objects are called *game-objects* and in ABS they are called *agents* but they are conceptually the same thing. The two main points Sweeney made were that dependent types could solve most of the run-time failures and that parallelism is the future for performance improvement in games. He distinguishes between pure functional algorithms which can be parallelized easily in a pure functional language and updating game-objects concurrently using software transactional memory (STM).

The thesis of [23] 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.

[24] and [25] 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.

VIII. 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 [7] 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

[6] Gregory [22] defines computer-games as *soft real-time interactive agent-based computer simulations*

[7] main author of the paper [16]

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

ACKNOWLEDGMENTS

The authors would like to thank I. Perez, H. Nilsson, P. Capriotti, J. Greensmith 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] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, Robots, and Functional Reactive Programming," in *Advanced Functional Programming*, ser. Lecture Notes in Computer Science, J. Jeuring and S. L. P. Jones, Eds. Springer Berlin Heidelberg, 2003, no. 2638, pp. 159–187, dOI: 10.1007/978-3-540-44833-4_6. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-44833-4_6

[12] A. Courtney, H. Nilsson, and J. Peterson, "The Yampa Arcade," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '03. New York, NY, USA: ACM, 2003, pp. 7–18. [Online]. Available: http://doi.acm.org/10.1145/871895.871897

[13] H. Nilsson, A. Courtney, and J. Peterson, "Functional Reactive Programming, Continued," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '02. New York, NY, USA: ACM, 2002, pp. 51–64. [Online]. Available: http://doi.acm.org/10.1145/581690.581695

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

[15] 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

[16] 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

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

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

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

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

[21] 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

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

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

[24] 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

[25] 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

APPENDIX A
FUNCTIONAL-REACTIVE CODE OF THE AGENT-BASED SIR MODEL

```
data SIRState = Susceptible | Infected | Recovered deriving (Eq)
data SIRMsg = Contact SIRState deriving (Eq)

type SIRAgentState = SIRState

type SIREnvironment = [AgentId]

type SIRAgentDef = AgentDef SIRAgentState SIRMsg SIREnvironment
type SIRAgentBehaviour = AgentBehaviour SIRAgentState SIRMsg SIREnvironment
type SIRAgentBehaviourReadEnv = ReactiveBehaviourReadEnv SIRAgentState SIRMsg SIREnvironment
type SIRAgentBehaviourIgnoreEnv = ReactiveBehaviourIgnoreEnv SIRAgentState SIRMsg SIREnvironment
type SIRAgentIn = AgentIn SIRAgentState SIRMsg SIREnvironment
type SIRAgentOut = AgentOut SIRAgentState SIRMsg SIREnvironment
type SIRAgentObservable = AgentObservable SIRAgentState

type SIREventSource = EventSource SIRAgentState SIRMsg SIREnvironment

-------------------------------------------------------------------------------
infectivity :: Double
infectivity = 0.05

contactRate :: Double
contactRate = 5

illnessDuration :: Double
illnessDuration = 15

contactSS :: Int
contactSS = 20

illnessTimeoutSS :: Int
illnessTimeoutSS = 2

-------------------------------------------------------------------------------
createSIRNumInfected :: Int -> Int -> IO ([SIRAgentDef], SIREnvironment)
createSIRNumInfected agentCount numInfected = do
    let agentIds = [0 .. (agentCount-1)]
    let infectedIds = take numInfected agentIds
    let susceptibleIds = drop numInfected agentIds

    adefsSusceptible <- mapM (sirAgent Susceptible) susceptibleIds
    adefsInfected <- mapM (sirAgent Infected) infectedIds

    return (adefsSusceptible ++ adefsInfected, agentIds)

sirAgent :: SIRState -> AgentId -> IO SIRAgentDef
sirAgent initS aid = do
    rng <- newStdGen
    let beh = sirAgentBehaviour rng initS
    let adef = AgentDef {
          adId = aid
        , adState = initS
        , adBeh = beh
        , adInitMessages = NoEvent
        , adConversation = Nothing
        , adRng = rng
        }

    return adef

-------------------------------------------------------------------------------
-- UTILITIES
gotInfected :: SIRAgentIn -> Rand StdGen Bool
gotInfected ain = onMessageM gotInfectedAux ain False
  where
    gotInfectedAux :: Bool -> AgentMessage SIRMsg -> Rand StdGen Bool
    gotInfectedAux False (_, Contact Infected) = randomBoolM infectivity
    gotInfectedAux x _ = return x

respondToContactWith :: SIRState -> SIRAgentIn -> SIRAgentOut -> SIRAgentOut
respondToContactWith state ain ao = onMessage respondToContactWithAux ain ao
```

```haskell
  where
    respondToContactWithAux :: AgentMessage SIRMsg -> SIRAgentOut -> SIRAgentOut
    respondToContactWithAux (senderId, Contact _) ao = sendMessage (senderId, Contact state) ao

-- SUSCEPTIBLE
sirAgentSuceptible :: RandomGen g => g -> SIRAgentBehaviour
sirAgentSuceptible g =
        transitionOnEvent
                sirAgentInfectedEvent
                (readEnv $ sirAgentSusceptibleBehaviour g)
                (sirAgentInfected g)

sirAgentInfectedEvent :: SIREventSource
sirAgentInfectedEvent = proc (ain, ao) -> do
    let (isInfected, ao') = agentRandom (gotInfected ain) ao
    infectionEvent <- edge -< isInfected
    returnA -< (ao', infectionEvent)

sirAgentSusceptibleBehaviour :: RandomGen g => g -> SIRAgentBehaviourReadEnv
sirAgentSusceptibleBehaviour g = proc (ain, e) -> do
    let ao = agentOutFromIn ain
    ao1 <- doOnce (setDomainState Susceptible) -< ao
    ao2 <- sendMessageOccasionallySrcSS
                    g
                    (1 / contactRate)
                    contactSS
                    (randomAgentIdMsgSource (Contact Susceptible) True) -< (ao1, e)
    returnA -< ao2

-- INFECTED
sirAgentInfected :: RandomGen g => g -> SIRAgentBehaviour
sirAgentInfected g =
        transitionAfterExpSS
                g
                illnessDuration
                illnessTimeoutSS
                (ignoreEnv $ sirAgentInfectedBehaviour g)
                sirAgentRecovered

sirAgentInfectedBehaviour :: RandomGen g => g -> SIRAgentBehaviourIgnoreEnv
sirAgentInfectedBehaviour g = proc ain -> do
    let ao = agentOutFromIn ain
    ao1 <- doOnce (setDomainState Infected) -< ao
    let ao2 = respondToContactWith Infected ain ao1
    returnA -< ao2

-- RECOVERED
sirAgentRecovered :: SIRAgentBehaviour
sirAgentRecovered = setDomainStateReact Recovered

-- INITIAL CASES
sirAgentBehaviour :: RandomGen g => g -> SIRState -> SIRAgentBehaviour
sirAgentBehaviour g Susceptible = sirAgentSuceptible g
sirAgentBehaviour g Infected = sirAgentInfected g
sirAgentBehaviour _ Recovered = sirAgentRecovered

------------------------------------------------------------------------------
runSIRSteps :: IO ()
runSIRSteps = do
    -- no collapsing/updating of environment
    params <- initSimulation updateStrategy Nothing Nothing shuffleAgents (Just rngSeed)
    (initAdefs, initEnv) <- createSIRNumInfected agentCount numInfected
    let dynamics = simulateAggregateTime initAdefs initEnv params dt t aggregate
    print dynamics

aggregate :: ([SIRAgentObservable], SIREnvironment) -> (Double, Double, Double)
aggregate (aobs, _) = (susceptibleCount, infectedCount, recoveredCount)
  where
    susceptibleCount = fromIntegral $ length $ filter ((Susceptible==) . snd) aobs
    infectedCount = fromIntegral $ length $ filter ((Infected==) . snd) aobs
    recoveredCount = fromIntegral $ length $ filter ((Recovered==) . snd) aobs
```

APPENDIX B
FUNCTIONAL-REACTIVE CODE OF THE SYSTEM-DYNAMICS SIR MODEL

```
totalPopulation :: Double
totalPopulation = 1000

infectivity :: Double
infectivity = 0.05

contactRate :: Double
contactRate = 5

avgIllnessDuration :: Double
avgIllnessDuration = 15

-- Hard-coded ids for stocks & flows interaction
susceptibleStockId :: StockId
susceptibleStockId = 0

infectiousStockId :: StockId
infectiousStockId = 1

recoveredStockId :: StockId
recoveredStockId = 2

infectionRateFlowId :: FlowId
infectionRateFlowId = 3

recoveryRateFlowId :: FlowId
recoveryRateFlowId = 4

--------------------------------------------------------------------------------
-- STOCKS
susceptibleStock :: Stock
susceptibleStock initValue = proc ain -> do
    let infectionRate = flowInFrom infectionRateFlowId ain

    stockValue <- (initValue+) ^<< integral -< (-infectionRate)

    let ao = agentOutFromIn ain
    let ao0 = setDomainState stockValue ao
    let ao1 = stockOutTo stockValue infectionRateFlowId ao0

    returnA -< ao1

infectiousStock :: Stock
infectiousStock initValue = proc ain -> do
    let infectionRate = flowInFrom infectionRateFlowId ain
    let recoveryRate = flowInFrom recoveryRateFlowId ain

    stockValue <- (initValue+) ^<< integral -< (infectionRate - recoveryRate)

    let ao = agentOutFromIn ain
    let ao0 = setDomainState stockValue ao
    let ao1 = stockOutTo stockValue infectionRateFlowId ao0
    let ao2 = stockOutTo stockValue recoveryRateFlowId ao1

    returnA -< ao2

recoveredStock :: Stock
recoveredStock initValue = proc ain -> do
    let recoveryRate = flowInFrom recoveryRateFlowId ain

    stockValue <- (initValue+) ^<< integral -< recoveryRate

    let ao = agentOutFromIn ain
    let ao' = setDomainState stockValue ao

    returnA -< ao'

--------------------------------------------------------------------------------
-- FLOWS
infectionRateFlow :: Flow
infectionRateFlow = proc ain -> do
```

```
    let susceptible = stockInFrom susceptibleStockId ain
    let infectious = stockInFrom infectiousStockId ain

    let flowValue = (infectious * contactRate * susceptible * infectivity) / totalPopulation

    let ao = agentOutFromIn ain
    let ao' = flowOutTo flowValue susceptibleStockId ao
    let ao'' = flowOutTo flowValue infectiousStockId ao'

    returnA -< ao''

recoveryRateFlow :: Flow
recoveryRateFlow = proc ain -> do
    let infectious = stockInFrom infectiousStockId ain

    let flowValue = infectious / avgIllnessDuration

    let ao = agentOutFromIn ain
    let ao' = flowOutTo flowValue infectiousStockId ao
    let ao'' = flowOutTo flowValue recoveredStockId ao'

    returnA -< ao''

-------------------------------------------------------------------------------
createSysDynSIR :: [SDDef]
createSysDynSIR =
    [ susStock
    , infStock
    , recStock
    , infRateFlow
    , recRateFlow
    ]
  where
    initialSusceptibleStockValue = totalPopulation - 1
    initialInfectiousStockValue = 1
    initialRecoveredStockValue = 0

    susStock = createStock susceptibleStockId initialSusceptibleStockValue susceptibleStock
    infStock = createStock infectiousStockId initialInfectiousStockValue infectiousStock
    recStock = createStock recoveredStockId initialRecoveredStockValue recoveredStock

    infRateFlow = createFlow infectionRateFlowId infectionRateFlow
    recRateFlow = createFlow recoveryRateFlowId recoveryRateFlow

-------------------------------------------------------------------------------
runSysDynSIRSteps :: IO ()
runSysDynSIRSteps = print dynamics
  where
    -- SD run completely deterministic, this is reflected also in the types of
    -- the createSysDynSIR and runSD functions which are pure functions
    sdDefs = createSysDynSIR
    sdObs = runSD sdDefs dt t

    dynamics = map calculateDynamics sdObs

-- NOTE: here we rely on the fact the we have exactly three stocks and sort them by their id to access them
--          stock id 0: Susceptible
--          stock id 1: Infectious
--          stock id 2: Recovered
--          the remaining items are the flows
calculateDynamics :: [SDObservable] -> (Double, Double, Double)
calculateDynamics unsortedStocks = (susceptibleCount, infectedCount, recoveredCount)
  where
    stocks = sortBy (\s1 s2 -> compare (fst s1) (fst s2)) unsortedStocks
    ((_, susceptibleCount) : (_, infectedCount) : (_, recoveredCount) : _) = stocks
```