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 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 which results in a surprisingly fresh, powerful and convenient EDSL for formulating ABS in Haskell.

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 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. Background

A. 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:

- 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.
- 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.

Fig. 1: Transitions in the SIR compartment model.

- 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

Functional Reactive Programming (FRP) is a way to implement systems with continuous and discrete time-semantics in pure functional languages. There are many different approaches and implementations but in our paper we use *arrowized* FRP as implemented in the library Yampa which we will introduce shortly. For an in-depth introduction into FRP using Yampa we refer to the papers of [3], [4] and [5]. The central concept in arrowized FRP is the Signal Function (SF) which can be understood as a mapping from an input- to an output-signal. A signal can be understood as a value which varies over time.

$$Signal\,\alpha \approx Time \to a$$
$$SF\,\alpha\,\beta \approx Signal\,\alpha \to Signal\,\beta$$

Signal functions are implemented as continuations which don't take a Δt at $t = 0$ but then change their signature into one which takes a Δt for $t > 0$. This allows to hide Δt completely from the types which makes them much more suitable for declarative programming. We also make use of Patersons do-notation for arrows [6] which makes the code much more readable. TODO: this section needs to be more refined but we cannot spend too much space on FRP as it is out of scope of the paper and we only need a rough understanding of the signal function concept because agents are implemented as signal functions.

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 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 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 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* [1].

[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.

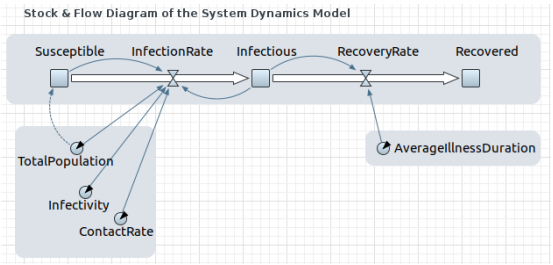


Fig. 2: A visual representation of the SD stocks and flows of the SIR compartment model. Picture taken using AnyLogic Personal Learning Edition 8.1.0.

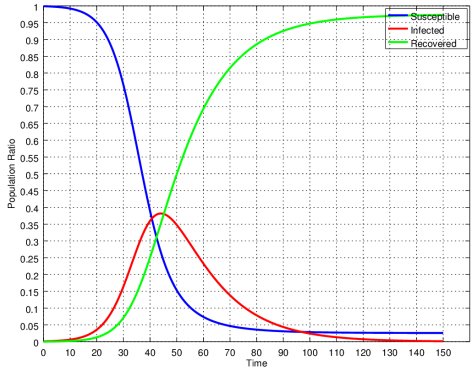


Fig. 3: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

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

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

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

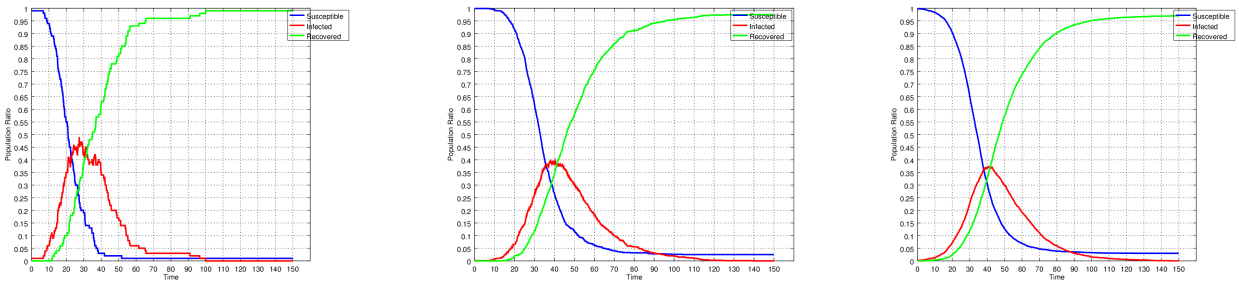
There exist a huge number of software packages which allow to conveniently express SD models using a visual approach like in Figure 2.

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

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 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 [7] pointed out.



(a) 100 Agents (b) 1,000 Agents (c) 5,000 Agents

Fig. 4: Approximating the continuous dynamics of the system dynamics simulation using the agent-based approach. Model-parameters are the same ($\beta = \frac{1}{5}$, $\gamma = 0.05$, $\delta = 15$ with initially 1 infected agent) except population size. All simulations run for 150 time-steps.

- 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 [7] with $\lambda = \frac{1}{\delta}$ and making the transition after this duration.

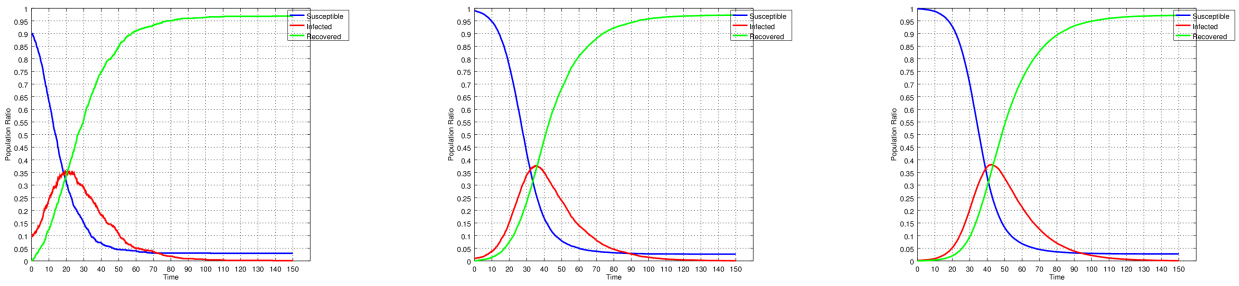
In Figure 4 we give the dynamics simulating the SIR model with the agent-based approach.

As previously mentioned the agent-based approach is a 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 5,000 Agents do not match the dynamics of the SD simulation perfectly. 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 10 replications, which matches the SD dynamics to a very satisfactory level [2].

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

We will derive the implementation of this approach in the next section, with the complete code provided in Appendix A. As will be seen our approach allows us express this behaviour very explicitly and is looking very much like a formal ABS specification of the problem.

[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 5,000 (which amounts to 0.2%) 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.



(a) 100 Agents (b) 1,000 Agents (c) 5,000 Agents

Fig. 5: Dynamics of Figure 4 averaged over 10 replications with initially 10 infected agents.

IV. Functional Reactive ABS

The challenges one faces when implementing an ABS plain, without support from a library are manifold. Generally one faces the following challenges:

- Agent Representation - how do we represent an agent in Haskell?
- Agent-Agent Interaction - how can agents interact with other agents in Haskell without resorting to the IO Monad?
- Environment representation - how can we represent an environment which must have the ability to update itself e.g. regrow some resources?
- Agent-Environment interaction - how can agents interact (read / write) with the environment?
- Agent Updating - how is the set of agents organised, how are they updated and how is it managed (deleting, adding during simulation) 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 A.

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 messages the agent understands
- **e** - the environment the agent 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.

Each agent may have a generic state which could be represented by any data type or compound data. A SIR agent's state can be represented using the an Algebraic Data Type (ADT) 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 [3]

```
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. Both types are opaque and access to them is only possible through the provided functions. 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 whereas *AgentOut* is both read- and write-able. The first thing an agent-behaviour does is creating the default AgentOut from the existing AgentIn as is done in line 94 in Appendix A.

```
agentOutFromIn :: AgentIn -> AgentOut
```

[3]Note that we omit the type-parameters in the following code-listings unless it is needed for clarity. Still it is important to keep in mind that all AgentIn and AgentOut are parameterised with *s* representing the type of its state, *m* representing the type of the messages and *e* representing the type of environment.

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:

```
type AgentId = Int

agentId :: AgentIn -> AgentId
kill :: AgentOut -> AgentOut
isDead :: AgentOut -> Bool
onStart :: (AgentOut -> AgentOut) -> AgentIn -> AgentOut -> AgentOut

agentState :: AgentOut -> s
setAgentState :: s -> AgentOut -> AgentOut
updateAgentState :: (s -> s) -> AgentOut -> AgentOut

createAgent :: AgentDef -> AgentOut -> AgentOut
```

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 *agentState* returns the agents' state *s*, *setAgentState* allows to change the state of the agent by overriding it and *updateAgentState* 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 -> AgentOut) -> SF AgentOut AgentOut
doOnceR :: AgentBehaviour -> AgentBehaviour
doNothing :: AgentBehaviour

setAgentStateR :: s -> AgentBehaviour
updateAgentStateR :: (s -> s) -> AgentBehaviour
```

The *doOnce* function may seem strange at first but allows conveniently make actions (which are changing the *AgentOut*) only once e.g. when making the transition from Susceptible to Infected changing the state to Infected just once as can be seen in line 114 of Appendix A. A more striking example would be to send a message just once after a transition. The *doOnceR* function is the reactive version, which allows to run an agent behaviour only once. 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 *AgentIn* to *AgentOut* 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 = SF (AgentIn, AgentOut) (AgentOut, Event ())
transitionOnEvent :: EventSource -> AgentBehaviour -> AgentBehaviour -> AgentBehaviour
```

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 *AgentIn* and *AgentOut* and returns a (potentially changed) *AgentOut* and the event upon to switch. This function is used 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 as can be seen in lines 81-90 in Appendix A.

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 which can be seen in line 106 in Appendix A:

```
transitionAfterExp :: RandomGen g => g -> Double -> AgentBehaviour -> AgentBehaviour -> AgentBehaviour
```

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 *messaging* with share-nothing semantics. In this case the agent sends messages which will arrive at the

[4] AgentDef simply contains the initial state, behaviour and id of the agent (amongst others). See line 50 - 57 in Appendix A.

receiver in the next step of the simulation, 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 another agent through *AgentOut* where incoming messages are queued in the *AgentIn* in unspecified order and can be processed when the agent is running the next time. The agent is free to ignore the messages and if it does not process them in the current step, 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 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. For messaging, 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 -> AgentOut
sendMessageTo :: AgentId -> m -> AgentOut -> AgentOut
sendMessages :: [AgentMessage m] -> AgentOut -> AgentOut
broadcastMessage :: m -> [AgentId] -> AgentOut -> AgentOut

hasMessage :: (Eq m) => m -> AgentIn -> Bool
onMessage :: (AgentMessage -> acc -> acc) -> AgentIn -> acc -> acc
onMessageFrom :: AgentId -> (AgentMessage -> acc -> acc) -> AgentIn -> acc -> acc
onMessageType :: (Eq m) => m -> (AgentMessage -> acc -> acc) -> AgentIn -> 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 implement the functionality of an infected agent which replies to an incoming *Contact* message with another *Contact Infected* message as can be seen in line 73 of Appendix A.

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 -> AgentBehaviour -> AgentBehaviour
```

Of course messaging sometimes may have specific time-semantics as in our SIR model. There susceptible agents make contact with β 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 m e = e -> AgentOut -> (AgentOut, AgentMessage m)

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

constMsgReceiverSource :: m -> AgentId -> MessageSource
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 *AgentOut* and environment and returns an *AgentOut*. This signal-function which performs the actual generating of the messages needs to be fed in the tuple but only returns the changed *AgentOut* 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 *AgentOut* and returns a tuple with a changed *AgentOut* 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 *AgentId* (omitting the sender True/False). The susceptible agent builds on this function to make contact with other agents as can be seen in line 96-100 of Appendix A.

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 same time-step or interact over multiple steps: agent A interacts with agent B where the semantics of the model need an immediate response from agent B - which can lead to further interactions initiated by agent A. An example would be negotiating a trading price between two agents to buy and sell goods between each other and then execute the trade. This must happen in the same time-step as constraints need to be considered which could be violated in asynchronous interactions. Basically the concept is always the same and is rooted in the fact that these interactions needs to transact in the current time-step 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. It 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. It is important to understand that *both* agents can change their state and the envirionment 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 read its *AgentIn* and manipulate its state. 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 AgentConversationReceiver s m e = AgentIn -> e -> AgentMessage m -> Maybe (s, m, e)
type AgentConversationSender m e = AgentOut -> e -> Maybe (AgentMessage m) -> (AgentOute, e)

conversation :: AgentMessage m -> AgentConversationSender -> AgentOut -> AgentOut
conversationEnd :: AgentOut -> AgentOut
```

The conversation 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). Note that conversations are inherently non-reactive because they are synchronous interactions, involve no time-semantics because time is halted, thus it makes no sense to implement conversations as signal-functions.

C. Environment representation

Agents have access to 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 [5].

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 the 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 *AgentId* 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.

[5]Exceptions in case when providing utility-functions which allow agents to operate on existing environment implementations

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 *AgentId* as the environment which means that every agent knows all the existing agents of the simulation and can address them - see line 6 in A. 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) identifies four possible ways of doing this where we only implemented the *sequential-* and *parallel-strategy* [6]. Implementing these iteration-strategies using Haskell and FRP is not as straight-forward as in imperative effectful languages 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 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 inelegant.

In both the sequential and parallel update-strategy each iteration must also output a potentially changed environment. As already discussed this is implemented as a signal-function which, when available, is then run after each iteration, to make the environment pro-active as well. An example of an environment behaviour would be to regrow some good on each cell according to some rate per time-unit.

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 *AgentIn* of the receiving agents. The environment which is passed in and returned as well, will then be passed forward to the next agent_{i+1} in the current iteration. The last environment is then the final environment in the current iteration and will be returned together with the current list of *AgentOut* (see below). 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 its 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 folded into a final one which is always domain-specific thus the model implementer needs to provide a corresponding function.

```
type EnvironmentFolding 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, the environment folding function is 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

[6]The other two strategies being the *concurrent-* and *actor-strategy*, both requiring to run within the STM-Monad, which is not possible with Yampa. Ivan Perez [9] 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.

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 in the given signal-function.

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

ignoreEnv :: ReactiveBehaviourIgnoreEnv -> AgentBehaviour
readEnv :: ReactiveBehaviourReadEnv -> AgentBehaviour
```

We use both functions in our SIR implementation. The function *readEnv* is used in line 83 of Appendix A to make sure the behaviour of a susceptible agent can read the environment but never change it. The function *ignoreEnv* is used in line 110 of Appendix A to make sure the behaviour of an infected agent never accesses the environment.

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 an imperative nature and map therefore naturally to a monadic style of programming using the *do* notation. Unfortunately it is not possible to do monadic programming within a signal-function, 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 -> AgentOut -> (AgentOut, e)) -> AgentBehaviour
agentPureReadEnv :: (e -> Double -> AgentIn -> AgentOut -> AgentOut) -> AgentBehaviour
agentPureIgnoreEnv :: (Double -> AgentIn -> AgentOut -> AgentOut) -> AgentBehaviour
```

The function *agentPure* wraps a non-monadic (pure) function and wraps it in a signal-function. This pure function has the environment, the current local time of the agent, the *AgentIn* and a default *AgentOut* as arguments 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 more restrictive version 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 -> State AgentOut e) -> AgentBehaviour
agentMonadicReadEnv :: (e -> Double -> AgentIn -> State AgentOut ()) -> AgentBehaviour
agentMonadicIgnoreEnv :: (Double -> AgentIn -> State AgentOut ()) -> AgentBehaviour
```

We also provide monadic versions of the pure primitives of our EDSL which work on *AgentOut* as discussed above. They run in the State Monad where the state is the *AgentOut* 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 ()
sendMessageToM :: AgentId -> m -> State AgentOut ()
onMessageM :: (Monad mon) => (acc -> AgentMessage m -> mon acc) -> AgentIn -> acc -> mon acc

createAgentM :: AgentDef -> State AgentOut ()

killM :: State AgentOut ()
isDeadM :: State AgentOut Bool

agentStateM :: State AgentOut s
updateAgentStateM :: (s -> s) -> State AgentOut ()
setAgentStateM :: s -> State AgentOut ()
agentStateFieldM :: (s -> t) -> State AgentOut t
```

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

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

G. Randomness

Most of the ABS are inherent stochastic processes and so is the agent-based SIR implementation. This means that agents behaviour depends on random-numbers. So far the drawing from these random-number was hidden behind functions but sometimes an agent needs to draw a random-number directly. For this we provide each agent with its own random-number generator which must be initialized when creating the agent-definition as seen in line 48-59 of Appendix A. The agent can then draw random-numbers in a pure way without having to resort to the IO Monad. Still it can become very cumbersome because

the changed random-number generator needs to be updated in the opaque *AgentOut* - for this we provide a few convenience functions and monadic implementations.

```
agentRandom :: Rand StdGen a -> AgentOut -> (a, AgentOut)
agentRandomM :: Rand StdGen a -> State AgentOut a

randomBoolM :: (RandomGen g) => Double -> Rand g Bool
```

Both functions *agentRandom* and *agentRandomM* allow to run a random action implemented in the Rand Monad acting on a StdGen generator. Both need an instance of an *AgentOut* which runs the random action on the agents random-number generator, updates it and returns the random value. The function *randomBoolM* is such a random action implemented in the Rand Monad and draws a random boolean which is True with a given probability. It is use to randomly determine if a susceptible agent got infected in case of a *Contact Infected* message as can be seen in line 67 of Appendix A. Note that this random action runs in *onMessageM* which allows to run on a generic monad. Note there exist more random action e.g. to pick at random from a list, to randomly generate within a range, to shuffle a list and to draw from a random distribution. We didn't discuss them here as they follow the same principle.

H. Running the Simulation

For actually running the simulation we provide three different approaches.

```
type AgentObservable s = (AgentId, s)
type SimulationStepOut s e = (Time, [AgentObservable s], e)
type AgentObservableAggregator s e a = SimulationStepOut s e -> a

simulateIOInit :: [AgentDef] -> e -> SimulationParams e
                  -> (ReactHandle () (SimulationStepOut s e) -> Bool -> SimulationStepOut s e -> IO Bool)
                  -> IO (ReactHandle () (SimulationStepOut s e))

simulateTime :: [AgentDef] -> e -> SimulationParams e -> DTime -> Time -> [SimulationStepOut s e]
simulateAggregateTime :: [AgentDef] -> e -> SimulationParams -> DTime -> Time -> AgentObservableAggregator a -> [a]
simulateTimeDeltas :: [AgentDef] -> e -> SimulationParams e -> [DTime] -> [SimulationStepOut s e]
```

The first one *simulateIOInit* allows for output in the IO Monad, which is useful when one wants to run the simulation for an undefined number of steps and visualise each step by rendering it to a window using a rendering library e.g. *Gloss* [7]. The *ReactHandle* is part of Yampas function *reactinit* and we refer to Yampas documentation for further details. The second approach *simulateTime* runs the simulation for a given time with given Δt and then returns the output for all Δt. The third approach *simulateTimeDeltas* works the same as the second one but one can provide a list of all Δt. The function *simulateAggregateTime* allows to transform the output of each step into a different representation, as happens in our SIR implementation where we aggregate the list of all observable agent-outputs into a tuple holding the number of susceptible, infected an recovered agents (see line 136 and 139 - 144 of Appendix A). In all approaches at every Δt we output a tuple of the current global simulation time, a list of *AgentId* with their states *s* and the environment *e*. All approaches take a list of initial agent definitions, the initial environment and simulation parameters which are obtained calling the function *initSimulation*:

```
data UpdateStrategy = Sequential | Parallel deriving (Eq)

initSimulation :: UpdateStrategy
                  -> Maybe (EnvironmentBehaviour e)
                  -> Maybe (EnvironmentFolding e)
                  -> Bool
                  -> Maybe Int
                  -> IO (SimulationParams e)
```

This function takes as parameters the update-strategy with which to run this simulation, an optional environment behaviour, an optional environment folding, a boolean which determines whether the agents are shuffled after each iteration [8] and an optional initial random-number generator seed.

I. Replications

As already described in the previous sections, sometimes it is necessary to run replications. This means running the same simulation multiple times but each with a different random-number generator and averaging the results. For this we provide replicators for agents and environments which can create a new *AgentDef* and environment *e* from the initial ones using the provided random-number generator for the current replication. Although it would be possible to use a default agent replicator which only replaces the random-number generator in *AgentDef*, in the case of our SIR implementation we also need to replace the behaviour signal-function which takes a random-number generator as well.

```
type AgentDefReplicator = StdGen -> AgentDef -> (AgentDef, StdGen)
type EnvironmentReplicator e = StdGen -> e -> (e, StdGen)
```

[7]Note that we provide substantial rendering functionality for the environments but don't discuss it here as it is out of the scope of the paper.

[8]This is necessary in some models which run in the sequential strategy to uniformly distribute the probability of an agent to run at a fixed position.

```
data ReplicationConfig e = ReplicationConfig {
    replCfgCount            :: Int,
    replCfgAgentReplicator  :: AgentDefReplicator,
    replCfgEnvReplicator    :: EnvironmentReplicator e
}

runReplications :: [AgentDef] -> e -> SimulationParams e -> DTime
                  -> Time -> ReplicationConfig s m e -> [[SimulationStepOut s e]]
```

The function *runReplications* works the same way as the previously mentioned *simulateTime* but takes an additional replication configuration and returns lists of *SimulationStepOut* of length *replCfgCount*.

V. Sampling the System

When sampling the system, the correct Δt must be selected which depends on the highest frequency which occurs in a time-reactive function in the whole system. For example in the SIR model we want infected agents to make on average contact with $\beta = 5$ other agents per time-unit, which means with a frequency of $\frac{1}{5}$. This functionality is built on Yampas function *occasionally* which behaviour we investigated under differing Δt with the above frequency. In this investigation we simply sampled occasionally with different Δt for a duration of $t = 1,000$ and the event-frequency of $\frac{1}{5}$. The result can be seen in Figure 6a and is quite striking. The plot clearly shows that occasionally needs a quite high sampling frequency even for a comparatively low event-frequency, which becomes of course worse for higher event-frequencies.

The other time-reactive function which occurs in the SIR model is the timed transition from infected to recovered which occurs on average with an exponential random-distribution after $\delta = 15$. 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 b* 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 Δt until the accumulator is greater of equal than the previously drawn exponential time-out. We also investigated the behaviour of this function under varying Δt using a time-out of $\delta = 15$. Our approach was to sample the *afterExp* until an event occurs [9] 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 6c. 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-sample the *afterExp* then we can be off by one Δt. If we under-sample *occasionally* we keep loosing events - the less difference between Δt and event-frequency, the more events we lose. Of course *afterExp* can also be used for very short time-outs e.g. $\frac{1}{5}$. We have investigated the behaviour of this function for various Δt as well as seen in Figure 6d. 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 6b. The result shows that for low frequencies occasionally works fine with larger Δt.

Super-Sampling

Of course performance is a big issue and it decreases as Δt get smaller and smaller. This is because when running a simulation for a duration of t and sampling it with Δt when the steps to calculate is $\frac{t}{\Delta t}$. In each step all agents are run, messages delivered and environments folded and updated which implies that the more steps the lower the performance. If we could perform super-sampling just for the given high-frequency functions with the whole system running in lower frequency then we could achieve a substantial performance boost. In Yampa there exists a function *embed* which allows to run a given signal-function with provided Δt but the problem is that this function does not really help because it does not return a signal-function. What we need is a signal-function which takes the number of super-samples *n*, the signal-function *sf* to sample and returns a new signal-function which performs super-sampling on it:

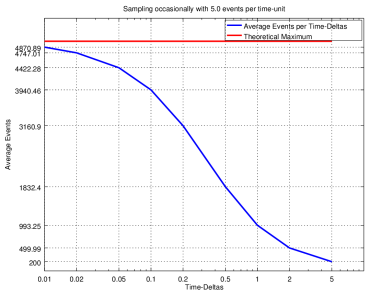
```
superSampling :: Int -> SF a b -> SF a [b]
```

It does this by evaluating *sf* for *n* times, each with $\Delta t = \frac{\Delta t}{n}$ 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.

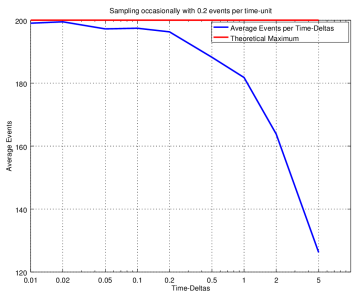
We ran tests super-sampling both *occasionally* Figure 7a, Figure 7b and *afterExp* Figure 7c, Figure 7d. They work the same way as above except that now $\Delta t = 1.0$ but using increasing numbers of super-samples. The results are as expected: as the number of super-samples increase, so increases 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

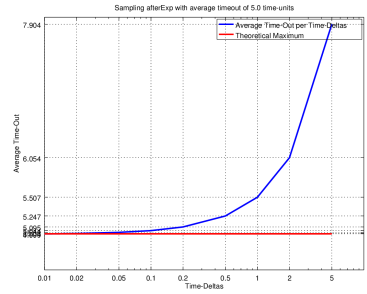
[9] This is one of the occasions where lazy evaluation really shines as one simply repeats the time-delta stream forever but then searches for the first occurrence of an event, which MUST occur at some point due to mathematical exponential distribution and our parameters to it, so it will always terminate.



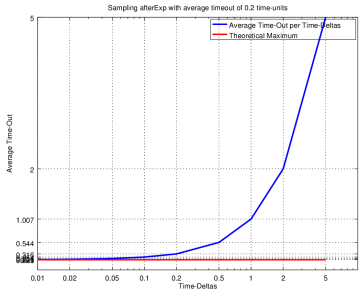
(a) Sampling *occasional* with a frequency of $\frac{1}{5}$ (average of 5 events per time-unit). The theoretical average is 5000 events within this time-frame.



(b) 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.



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



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

Fig. 6: Sampling the *afterExp* and *occasional* functions to visualise the influence of sampling frequencies on the occurrence of the respective events. Δt 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 $t = 1,000$ with 100 replications.

low frequency of $\Delta t = 1.0$. So we are then increasing the sampling-frequency just where we need it and keep the frequency low where it is not required.

Super-Sampling in SIR

We are using super-sampling in our SIR implementation to increase performance. We do this by setting $\Delta t = 1.0$ 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:

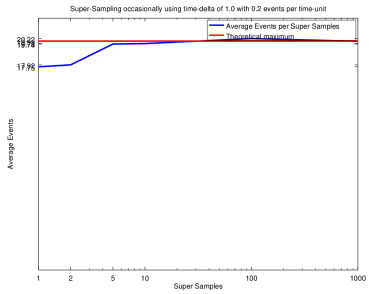
```
sendMessageOccasionallySrcSS :: RandomGen g => g -> Double -> Int -> MessageSource
                             -> SF (AgentOut, e) AgentOut

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

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 $\Delta t = 1.0$ 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. So in the end it is a struggle of performance vs. sufficiently close approximation. We define the number of super-samples in lines 29 and 32 and use the functions in line 96 and 106 of Appendix A.

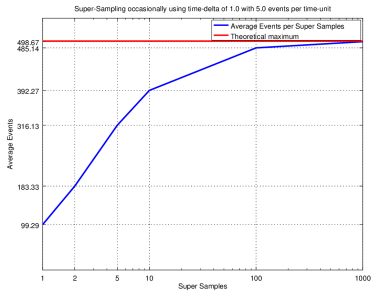
Message Round-Trips

Unfortunately when setting $\Delta t = 1.0$ the dynamics of the agent-based approach won't approach the dynamics of the SD, despite using super-sampling as can be seen in Figure 8a.

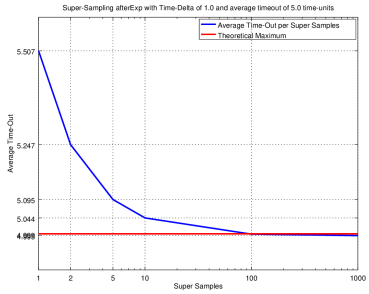
When reflecting on the messaging mechanism it becomes apparent that a round-trip from sender to receiver and back takes $2\Delta t$. A round-trip happens in our agent-based SIR approach to implement the transition from infected to susceptible - susceptible agents send *Contact Susceptible* messages to random agents (except itself) where only infected agents reply



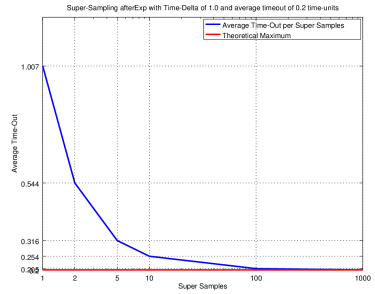
(a) Super-Sampling the *occasional* function with event-frequency of 5 (average of 0.2 events per time-unit). The theoretical average is 20 event within this time-frame.



(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.

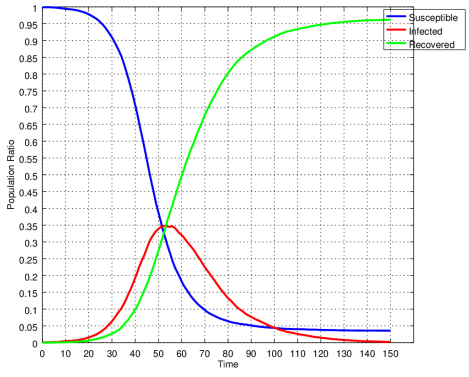


(c) Super-Sampling the *afterExp* function with average time-out of 5.

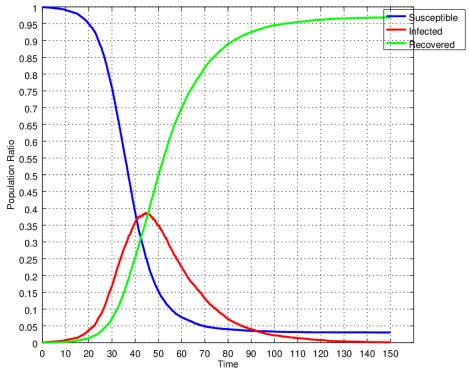


(d) Super-Sampling the *afterExp* function with average time-out of 0.2.

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 $\Delta t = 1.0$ 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 $t = 100$ with 100 replications.

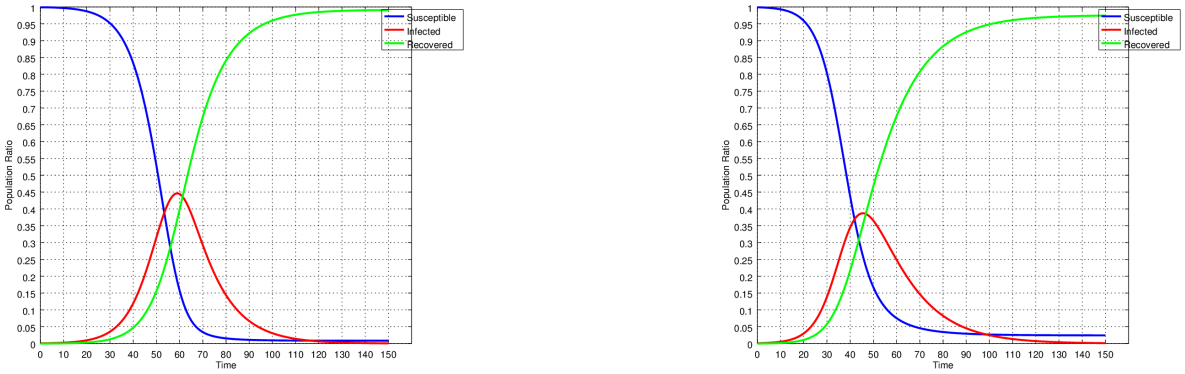


(a) $\Delta t = 1.0$



(b) $\Delta t = 0.1$

Fig. 8: Comparing the influence of different Δt. Both dynamics were generated with the same configuration of 10,000 agents, super-sampling enabled as described and the same model-parameters. When using $\Delta t = 1.0$, the dynamics do not match the ones of the SD approach, whereas in the case of $\Delta t = 0.1$, they can be seen as matching completely.



(a) $\Delta t = 1.0$ (b) $\Delta t = 0.1$

Fig. 9: Simulating the SIR model with our SD emulation using different Δt. Note that although $\Delta t = 0.1$ might seem very close the system dynamic solution, there are still subtle differences to the initial Figure 3 which uses $\Delta t = 0.01$.

with a *Contact Infected* message. This means that it takes $2\Delta t$ until a susceptible agent might get infected. This becomes an issue if we want to match the dynamics of our agent-based approach to the one of SD in which no time-delay happens - the agents act instantaneous with each other during one time-step. We have two solutions for this problem: either we resort to *conversations* or we increase the global sampling frequency of the system which matches the *message frequency* of messages which are subject to round-trips. Implementing conversations is only available in the *sequential* update-strategy and is much more involved, so we followed the approach of increasing the frequency. As can be seen in Figure 8b when setting $t\Delta = 0.1$ the resulting dynamics are a sufficiently good approximation to the SD solution.

VI. Results and Discussion

A. Emulating System Dynamics

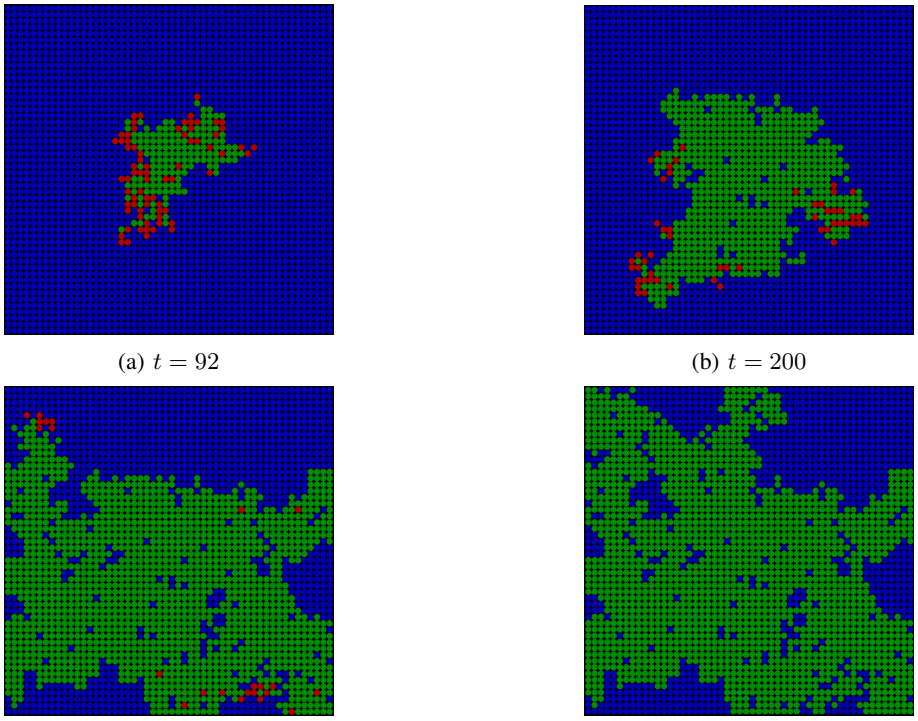
Due to the continuous time-semantics which can be expressed in our agent-based approach we can also emulate SD. Every stock and flow is then just an agent which exchange messages where the simulation is stepped using the *parallel* update-strategy. We add wrappers and type-definitions for convenience which increases the expressivity of the code, resembling system dynamics specifications. As a proof-of-concept we emulated the system dynamics of the SIR model, the code can be seen in Appendix B. Note that the code really looks like a SD specification with the integrals of the mathematical specification directly showing up in the code - the implementation is correct per definition. Due to the internal implementation of the *integral* function of Yampa which uses the rectangle-rule to integrate, one must ensure to sample the system dynamics with small enough Δt as becomes apparent in Figure 9.

B. Spatiality and Networks

When emulating the dynamics of the SIR model using an agent-based approach the question arises what we ultimately gain from doing so when we could have generated the dynamics much quicker and smoother using the SD approach. The difference is that the agent-based approach is a stochastic one and can thus also generate ”degenerated” dynamics e.g. in which the disease dies out after a few steps or even can’t spread from patient zero - in this case ABS is clearly a benefit as it allows to investigate *alternative futures*, something not possible with SD in which the disease will never die out prematurely when there are non-zero infected agents.

Another advantage of ABS over the system-dynamics approach is that agents can be heterogeneous and make use of spatial- and/or network-information defining the neighbourhood. We can thus simulate the spread of the disease throughout a population which is laid out on a 2D grid or one can investigate spreading of the disease throughout a network of agents where some are vaccinated and others not. We provide already suitable environments to simulate these cases and show an example of spreading the disease on a 2D grid in Figure 10.

When using a 2D grid or network one needs to set them up in the initialization code so there is a little more work to do there but the implementation of the agents differ just in one single line, which is where the neighbourhood is picked (see line 100 of Appendix A). Instead of *randomAgentIdMsgSource* one uses either *randomNeighbourNodeMsgSource* in the case of a network or *randomNeighbourCellMsgSource* in case of a 2D grid.



(a) $t = 92$ (b) $t = 200$

(c) $t = 440$ (d) $t = 873$

Fig. 10: Simulating SIR on a 52x52 grid with Moore neighbourhood using $\Delta t = 1$. Blue are susceptible, red are infected, green are recovered. The green areas act as protection as infected cannot cross the recovered border: this is particularly visible in the lower right corner of 10c where the disease has been contained in the blue island and has no means to escape. It may seem that the few remaining infected agents in the top left corner of 10c will die out soon but still it needs more than the already running simulation-time until the disease actually dies out with the last patient recovering at center top of 10d at $t = 873$.

C. Agents as Signals

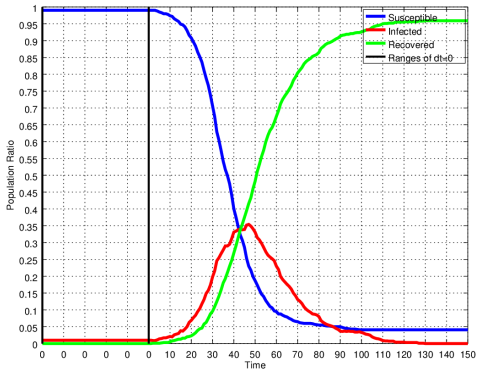
Due to the underlying nature and motivation of FRP and Yampa, agents can be seen as signals which are generated and consumed by a signal-function which is the behaviour of an agent. If an agent does not change, the output-signal should be constant amd if the agent changes e.g. by sending a message, changing its state,... the output-signal should change as well. A dead agent then should have no signal at all. The question is if the agents of our agent-based SIR implementation are true signals: do the dynamics stay constant when we sample the system with $\Delta t = 0$? We hypothesize that our agents are true signals, thus they should not change when time does not change because they are completely time-dependent and rely completely on time-semantics. When actually running the simulation with $\Delta t = 0$ one gets the results as seen in Figure 11.

As can be seen the dynamics are becoming constant *but* with a minor delay: infected increases a bit while susceptible decreases as can be seen in Figure 12. This is due to the delay of message delivery which takes one Δt, independent of its value - messages are also delivered when $\Delta t = 0$. Only message-generating functions, which depend on non-zero Δt to generate messages, will then stop generating messages. Reactive functions which act on incoming messages can still create change as they do not rely on time-semantics but just on the discrete event of a message arrival - which is the case in the transition from susceptible to infected.

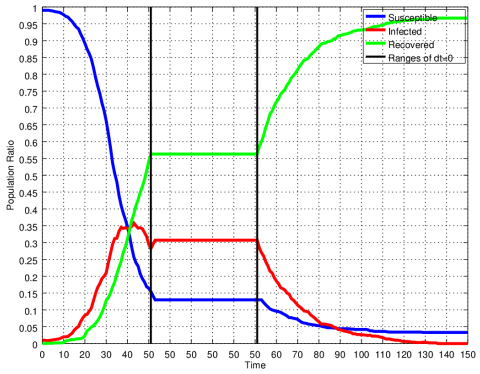
Note that agents of models with no time-semantics won't exhibit this behaviour - the dynamics will change even in case of $\Delta t = 0$ as agents act on every update and don't care about Δt and just assume that every update occurs after Δt independent of the actual value of it. We implemented the function *doRepeatedlyEvery* which allows to transform a time-agnostic agent-behaviour into one. It is built on Yampas *repeatedly* function and has the following signature:

```
doRepeatedlyEvery :: Time -> AgentBehaviour -> AgentBehaviour
```

This function takes a time interval and an agent behaviour signal-function and returns a new agent behaviour signal-function which runs the argument signal-function every time-interval. Note that this function is subject to sampling issues too e.g. when the time-interval is very small one needs to run the simulation with a $\Delta t \leq Time$ otherwise the dynamics would show delayed activation of the agent behaviour.



(a) $\Delta t = 0$ from step 0 to 50.



(b) $\Delta t = 0$ from step 51 to 101.

Fig. 11: Dynamics of agent-based SIR implementation of 1,000 agents running with $\Delta t = 1$ with ranges of $\Delta t = 0$ marked with two vertical black lines.

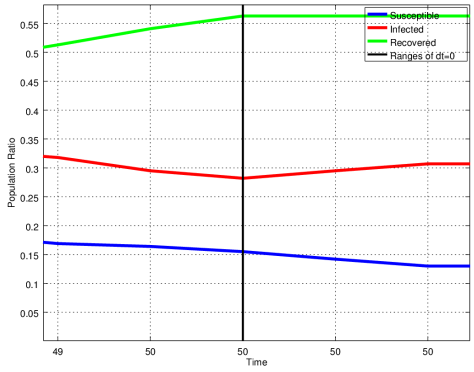


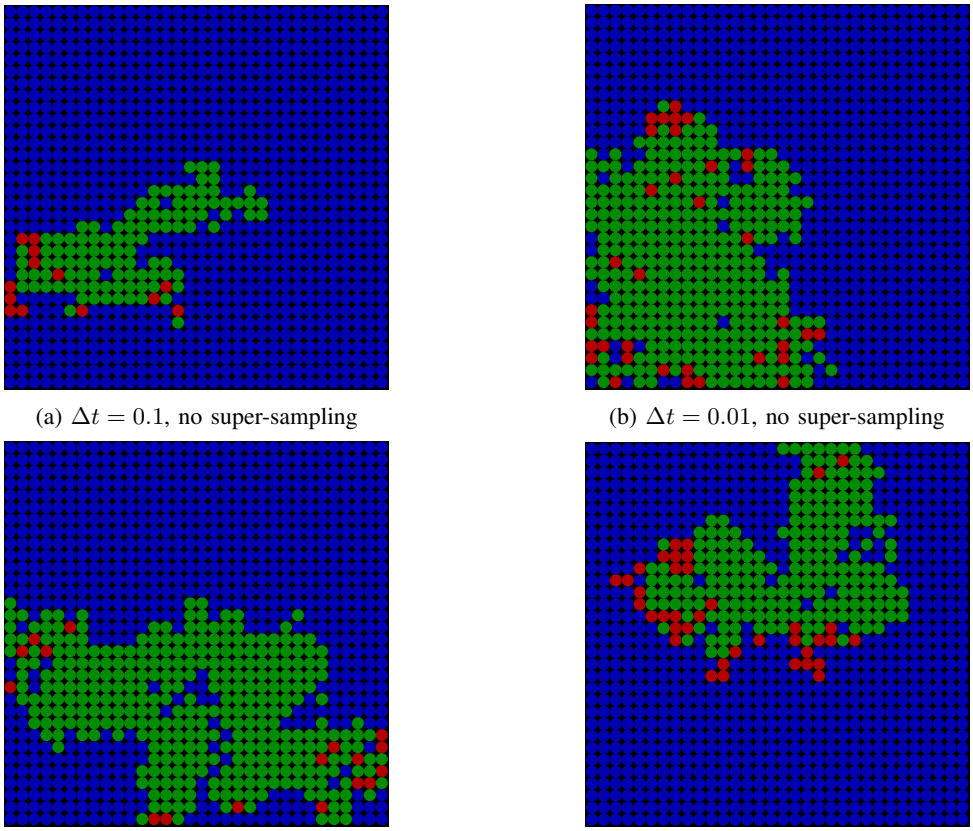
Fig. 12: Zoom-in to step 51, marked with the black line from where on $\Delta t = 0$ for the next 50 steps. The recovered ratio stays constant but a few agents get infected even *after* having switched to $\Delta t = 0$ which happens due to the message delivery lag. After all messages have been delivered, the signal stays constant until non-zero Δt are turned on again.

D. Randomness

It is important to note that if we disable super-sampling and run the simulation for a given time t but with two different Δt we would end up with two different results, even if the Δt are small enough to sample the time-dependent functions sufficiently. Also if we use super-sampling with $\Delta t = 1.0$ and create the exact same number of samples as when using no super-sampling but smaller Δt, then we also end up with different results. The reason for this behaviour is that most of the time-dependent functions ultimately build upon drawing from random-distributions. With different Δt we are generating a different number of random-samples, which would result in different random-number sequences which in turn ultimately leads to slightly different dynamics. When generating a plot of the dynamics this is not as visible, also this is the reason why one generates multiple replications, but this behaviour becomes strikingly apparent when simulating the SIR model on a 2D grid as can be seen in Figure 13.

E. Recursive ABS

Due to the inherent recursive nature of functional programming we came up with the idea of *recursive* ABS in which agents can recursively run the simulation within the simulation which would allow them to project their own actions into the future. So far it only exists as a proof-of-concept and we are currently only aware of a single model [10] in the field of ABS which does recursive simulation. The implementation of recursive ABS is very natural due to the explicit data-flow and lack of side-effects which eases the task very much. Unfortunately we cannot go into detail of our approach as it is beyond the scope of the paper.



(a) $\Delta t = 0.1$, no super-sampling

(b) $\Delta t = 0.01$, no super-sampling

(c) $\Delta t = 1.0$ with 10 super-samples

(d) $\Delta t = 1.0$ with 100 super-samples

Fig. 13: Comparing results on 32x32 grid after $t = 200$ but with different Δt and different number of super-samples.

F. Advantages

We now look at a number of advantages this functional reactive approach to ABS has over to the traditional imperative, object-oriented implementation approaches done in Phyton, Java, C++.

1) Continuous Time: It seems that in our approach we combine the benefits of SD and ABS: we have continuous time-semantics but with individual, heterogenous agents.

2) Code close to specification: When looking at the code of the agent-based implementation in Appendix A and SD implementation in Appendix B, both look very much like a specification. By creating this EDSL which allows to express powerful time-semantics it is possible to now create an ABS in a declarative style in Haskell where the agent-implementation looks very much like a model-specification thus being correct by definition. We claim that this is not only true for models with time-semantics but also with models which lack time-semantics and resemble a more imperative style of behaviour. We can also capture this using monadic programming using the State Monad for which we provide EDSL primitives as well which support all necessary operations in a monadic context.

3) Being pure: Because no part of the simulation runs in the IO monad [10] and we do not use unsafePerformIO [11] we can rule out a serious class of bugs caused by implicit data-dependencies and side-effects which can occur in traditional imperative implementations. Also we can statically guarantee the reproducibility of the simulation. Within the agents there are no side effects possible which could result in differences between same runs (e.g. file access, networking, threading, random-number seeding). Every agent has access to its own random-number generator, allowing randomness to occur in the simulation but the random-generator seed is fixed in the beginning and can never be changed within an agent to come from e.g. the current system time, which would require to run within the IO Monad. This means that after initialising the agents, which *could*

[10]Except when using graphical rendering, but then only the rendering happens in the IO Monad - the simulation step is always a pure computation

[11]We actually do use it when one wants to generate unique agent-ids for new agents which are created during the simulation. This is necessary because in this case we need to guarantee that two invocations will result in two different ids, which would be difficult / impossible when running the simulation in the *parallel* update-strategy. This is not a problem as long as an agent does not rely on the absolute value of an agent-id but just uses it as an opaque identifier for messaging.

run in the IO Monad, the simulation itself runs completely deterministic. We provide functionality to render the output to a window (using the Gloss library) or writing to a text-file, meaning, parts of the simulation would run in the IO Monad. Here we rely on Yampas *reactimate* function which provides a well-defined way of communicating with the world in such a system. This function provides the Δt for the next step, which *could* come from IO Monad but we forbid this and keep the Δt always fixed, thus removing another source of non-reproducibility where Δt is influenced by sytem dependent non deterministic rendering-performance every step as happens in games or described by [11] in the context of FRP.

4) Robust time handling: The actual Δt is never visible leads to an even more declarative style and supports the EDSL greatly. This also makes it impossible to mess around with time.

5) Replications: We nearly get replications for free without having to worry about side-effects and can even run them in parallel without headaches.

G. Disadvantages

We identify two main disadvantages.

1) Performance: Performance is currently no where near imperative object-oriented implementations. The reason for this is that we don't have in-place updates of data-structures and make no use of references. This results in lots of copying which is simply not necessary in the imperative languages with implicit effects. Also it is much more difficult to reason about time and space in our approach. Thus we see performance clearly as the main drawback of the functional approach and the only real advantage of the imperative approach over our.

2) Steep learning curve: Our approach is quite advanced in three ways. First it builds on the already quite involved FRP paradigm. Second it forces one to think properly of time-semantics of the model, how to sample it, how small Δt should be and whether one needs super-sampling or not and if yes how many samples one should take. Third it requires to think about agent-interaction and update-strategies, whether one should use conversations which forces one to run sequentially or if one can run agents in parallel and use normal messaging which incurs a time-delay which in turn would need to be considered when setting the Δt.

VII. Equivalence of SD and ABS implementation

After having shown that both the SD and ABS approaches are equivalent by visually comparing the dynamics of Figure 3 to the ones of Figure 8b, we ask the question whether we can also show the equivalence of both approaches from their implementation. First we need to show that the SD implementation is correct as in *is an implementation of the mathematical definition* as it is the foundation upon we build our equivalence.

A. Correctness of SD implementation

The mathematical formulas for susceptible, infected an recovered stocks with the infection- and recover-rates are directly translated into the SD implementation as seen in Appendix B. We can thus assume that the translation from the mathematical definition to Haskell code is correct as it *is* exactly the same. This leaves us with the question how the values of the stocks and flows are distributed between each other. When solving the mathematical equations numerically, one is dividing time into infinitely small intervals and calculates the new value of each formula at each time-interval at the same time: the values update all at the same time. The same needs to happen in our SD implementation which is indeed the case: *runSD* as seen in line 127 uses *simulateTime* behind the scenes with the *parallel* update-strategy which uses a *map* to iterate over all agents. This implies by definition of map that all stocks and flows, which are in fact agents, update virtually at the same time. Due to pureness of runSD we can rule out side-effects which could only occur when running in the IO Monad or using unsafePerformIO [12]. It is clear now that the values update at the same time and calculate the correct values as defined in the mathematical formulas. The distribution of the values happens by using absolute stock- and flow-ids and it is easy to check that the ids used in *flowInFrom/flowOutTo* and *stockInFrom/stockOutTo* build the correct connections as seen in the Stock-And-Flow diagram in Figure 2. This leaves us with the implementation of *integral*. Theoretically we arrive at the correct mathematical solution if we use infinitely small Δt but this is of course impossible using a computer. When looking at the implementation of *integral* it becomes apparent that it uses the rectangle-rule. TODO: need more details and look a bit into theory of numerically solving integral (e.g. classic newton, or runge-kutta). The rectangle rule needs small Δt for sufficient accuracy as we have shown in Figure 9. For our reference dynamics in Figure 3 we used $\Delta t = 0.001$, which computes 150,000 steps. When comparing it with dynamics as seen in Figure 14 generated by the professional software package AnyLogic Personal Learning Edition 8.1.0 we can safely say that a $\Delta t = 0.001$ is sufficiently small to generate matching accuracy.

[12]Again we use unsafePerformIO in initializing the previously mentioned agent-id generator but it is never used within the runSD implementation or any of the stocks and flows.

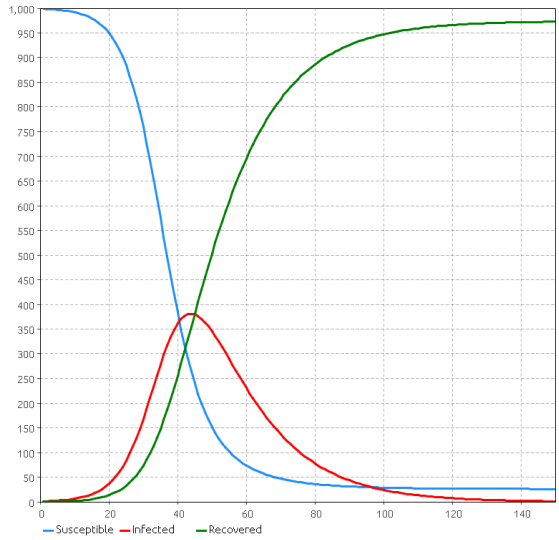


Fig. 14: Dynamics of the SIR model using the SD approach generated with AnyLogic Personal Learning Edition 8.1.0. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

B. Correctness of ABS implementation

The question is now what *correctness* of an ABS implementation means. This is indeed a very non-trivial problem and is highly dependent on the model. In our case of the agent-based SIR implementation we define correctness to mean *qualitatively approximating the SD dynamics*. This means that to show that our ABS implementation is correct, we need to show that it is equivalent to the SD implementation. TODO: this is a bit short, maybe we can say more about this. need to read a few papers and get better understanding on this topic

C. Equivalence

TODO: need to think about this very deeply, but basically it is all about probabilities which increase with number of messages which increase with number of infected. we need somehow to match the number of messages a susceptible receives to the stocks and flows formulas TODO: can we show why the infected themselves do not (and must not) make contact and only reply to incoming contacts?

VIII. Related Research

The amount of research on using the pure functional paradigm using Haskell in the field of ABS has been moderate so far. Most of the papers look into how agents can be specified using the belief-desire-intention paradigm [12], [13], [14]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [15]. It comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. The authors of [14] discuss using functional programming for DES and explicitly mention the paradigm of FRP to be very suitable to DES. In his talk [16], Tim Sweeney CTO of Epic Games discussed programming languages in the development of game-engines and scripting of game-logic. Although the fields of games and ABS seem to be very different, in the end they have also very important similarities: both are simulations which perform numerical computations and update objects in a loop either concurrently or sequential [13]. 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 [18] 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.

[13] Gregory [17] defines computer-games as ”soft real-time interactive agent-based computer simulations”.

The authors of [19] and [20] 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 which they claim is also readable. This supports our initial claim that FRP is a promising approach to implement ABS in Haskell.

IX. Conclusion and future research

In this paper we presented a novel approach to implement agent-based models by Functional Reactive Programming (FRP) in Haskell building on the library Yampa which we termed Functional Reactive Agent-Based Simulation [14].

In general, FRP tries to shift the direction of data-flow, from message passing onto data dependency. This helps reason about what things are over time, as opposed to how changes propagate TODO: cite ivan perez phd thesis. This of course raises the question whether FRP is *really* the right approach, because in our approach to ABS, messaging 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 messaging 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 messaging.

In the paper, TODO: cite my own art of iterating paper, the authors discuss four update-strategies of which we implemented only the *sequential* and *parallel* ones. The other two strategies are inherent concurrent ones, thus to implement them would require the agents to use concurrency features like STM or run in the IO Monad. Unfortunately both require monadic code which is not possible in Yampa. In [9] the authors provide a generalisation to FRP which also allows to run monads in FRP, which would make running the agents in the STM Monad possible. Using this approach we could then implement true concurrent agents allowing to implement the other two update-strategies. It would be interesting to see the benefits and drawbacks of our approach in these two strategies, especially when treating agents like signals as in the case of our SIR implementation. We leave this for further research.

Functional Programming is particularly strong for reasoning about an implementation and verification of an implemented model. How we can test, reason about and verify implemented models of ABS in our apprach, we leave for further research in an accompanying paper which will be written for the agent-based simulation audience.

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] 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

[4] 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

[5] 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

[6] R. Paterson, "A New Notation for Arrows," in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '01. New York, NY, USA: ACM, 2001, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/507635.507664

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

[8] 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

[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

[10] J. B. Gilmer, Jr. and F. J. Sullivan, "Recursive Simulation to Aid Models of Decision Making," in *Proceedings of the 32Nd Conference on Winter Simulation*, ser. WSC '00. San Diego, CA, USA: Society for Computer Simulation International, 2000, pp. 958–963. [Online]. Available: http://dl.acm.org/citation.cfm?id=510378.510515

[11] I. Perez and H. Nilsson, "Testing and Debugging Functional Reactive Programming," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 2:1–2:27, Aug. 2017. [Online]. Available: http://doi.acm.org/10.1145/3110246

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

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

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

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

[16] 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

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

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

[14] For this research implemented a prototype library called *FrABS* which is available under https://github.com/thalerjonathan/phd/tree/master/coding/libraries/frABS. We plan on releasing it on Hackage in the future.

[19] 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

[20] 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
```

```haskell
respondToContactWith :: SIRState -> SIRAgentIn -> SIRAgentOut -> SIRAgentOut
respondToContactWith state ain ao = onMessage respondToContactWithAux ain ao
  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 \label{line_agentOutFromIn}
    ao1 <- doOnce (setAgentState 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 (setAgentState Infected) -< ao
    let ao2 = respondToContactWith Infected ain ao1
    returnA -< ao2

-- RECOVERED
sirAgentRecovered :: SIRAgentBehaviour
sirAgentRecovered = setAgentStateR Recovered

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

--------------------------------------------------------------------------------
runSIR :: IO ()
runSIR = 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 :: (Time, [SIRAgentObservable], SIREnvironment) -> (Time, Double, Double, Double)
aggregate (t, aobs, _) = (t, 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 = setAgentState 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 = setAgentState 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' = setAgentState stockValue ao

    returnA -< ao'
```

```haskell
-------------------------------------------------------------------------------
-- 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 :: (Time, [SDObservable]) -> (Time, Double, Double, Double)
calculateDynamics (t, unsortedStocks) = (t, susceptibleCount, infectedCount, recoveredCount)
  where
    stocks = sortBy (\s1 s2 -> compare (fst s1) (fst s2)) unsortedStocks
    ((_, susceptibleCount) : (_, infectedCount) : (_, recoveredCount) : _) = stocks
```