Functional Reactive Agent-Based Simulation

Towards pure functional 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

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

***Abstract*—So far, the pure functional paradigm hasn't got much attention in Agent-Based Simulation (ABS) where the dominant programming paradigm is object-orientation, with Java, Python and C++ being its most prominent representatives. We claim that pure functional programming using Haskell is very well suited to implement complex, real-world agent-based models and brings with it a number of benefits. To show that we implemented the library *FrABS* which allows to do ABS the first time in the pure functional programming language Haskell. To achieve this we leverage the basic concepts of ABS with functional reactive programming using Yampa. The result is a surprisingly fresh approach to Agent-ABS as it allows to incorporate discrete time-semantics similar to Discrete Event Simulation and continuous time-flows like System Dynamics. In this paper we will show the novel approach of FrABMS through the example of the SIR model, discuss implications, benefits and best practices.**

***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 constituting parts of the system, called *Agents* and their local interactions. From these local interactions then the emergent property of the system emerges. ABS is still a young field, having emerged in the early to mid 90s primarily in the fields of social simulation and computational economics.

The authors of the seminal Sugarscape model [1] explicitly advocate object-oriented programming as "a particularly natural development environment for Sugarscape specifically and artificial societies generally." and report about 20.000 lines of code which includes GUI, graphs and plotting. They implemented their simulation software in Object Pascal and C where they used the former for programming the agents and the latter for low-level graphics [2]. Axelrod [3] recommends Java for experienced programmers and Visual Basic for beginners. Up until now most of ABS seems to have followed this suggestion and are implemented using programming languages which follow the object-oriented imperative paradigm.

A serious problem of object-oriented implementations is the blurring of the fundamental difference between agent and object - an agent is first of all a metaphor and *not* an object. In object-oriented programming this distinction is obviously lost as in such languages agents are implemented as objects which leads to the inherent problem that one automatically reasons about agents in a way as they were objects - agents have indeed become objects in this case. The most notable difference between an agent and an object is that the latter one do not encapsulate behaviour activation [4] - it is passive. Also it is remarkable that [4] a paper from 1999 claims that object-orientation is not well suited for modelling complex systems because objects behaviour is too fine granular and method invocation a too primitive mechanism.

As ABS is almost always used for scientific research, producing often break-through scientific results as pointed out in [5], these ABS need to be *free of bugs*, *verified against their specification*, *validated against hypotheses* and ultimately be *reproducible*. One of the biggest challenges in ABS is the one of validation. In this process one needs to connect the results and dynamics of the simulation to initial hypotheses e.g. *are the emergent properties the ones anticipated? if it is completely different why?*. It is important to understand that we always *must have* a hypothesis regarding the outcome of the simulation, otherwise we leave the path of scientific discovery. We must admit that sometimes it is extremely hard to anticipate *emergent patterns* but still there must be *some* hypothesis regarding the dynamics of the simulation otherwise we drift off into guesswork.

In this paper we ask how ABS can be done using the pure functional programming paradigm using Haskell, what the benefits are and if it could overcome the critique of using object-orientation in this field.

II. Background

[ ] time in FrABS: when 0dt then still actions can occur when not relying on time semantics [ ] what about time-travel in abms for introspection during running it? this is much easier in FrABS

TODO: reasoning about dynamics in code would allow us to cut substantial calculations: can make assumptions about dynamics without actually running it. is it even possible?

TODO: property-based and unit testing of a model

TODO: modular testing of agents

TODO: also solve the SIR with an algebra system to have a bullet-proof "proof" that we reproduce the same dynamics. this is only partially a proof our system is correct, but it is not a formal proof, this needs to be done different

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

TODO: publishing: 1st version write for a journal in the ABMS community, 2nd version write for a conference in the functional programming community e.g. for the TFP in Kent 2018

TODO: It should be possible to formally show that spatial SIR and WildFire are the same model. NOTE: they are NOT the same, the fundamental difference is that in the WildFire model only the burning cells initiate the ignition - if we compare this to the SIR, the burning cells would be infected agents and although in the spatial SIR model the infected agents make contact with other agents, so do the susceptible ones which does NOT occur in wildfire

TODO: cite my own work on update-strategies

TODO: can we formally show that the SIR approximates the SD model?

TODO: cite papers which discuss how to approximate a SD model by ABS - Macal (2010) - To Agent-Based Simulation From System Dynamics -¿ i am very unhappy with this paper: first it does not give concrete parameters for the SD model so it is impossible to replicate. Also i think it has a systematical error as the infected agents make no contact but this is required as evident from the SD-models infection-rate which also incorporates. TODO: write an email to this guy: why are the infectious not contacting the other agents? this seems to be a systematical error - Borshchev, Filippov (2004) - From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools -¿ its VERY IMPORTANT point is that we need to draw the illness-duration from an exponential-distribution because the illness-duration is proportional to the size of the infected. note: this is wrongly expressed, need to find the correct formulation

-¿ my emulation of SD using ABS is really an implementation of the SD model and follows it - they are equivalent -¿ my ABS implementation is the same as / equivalent to the SD emulation =¿ thus if i can show that my SD emulation is equlas to the SD model =¿ AND that the ABS implementation is the same as the SD emulation =¿ THEN the ABS implementation is an SD implementation, and we have shown this in code for the first time in ABS

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

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

A. Agent Based Simulation

We understand ABS as a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is of knowledge. Those parts, called agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. So the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in a network of neighbours by exchange of messages [6]. It is important to note that we focus our understanding of ABS on a very specific kind of agents where the focus is on communicating entities with individual, localized behaviour from out of which the global behaviour of the system emerges. We informally assume the following about our agents:

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

Epstein [7] identifies ABS to be especially applicable for analyzing *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. Thus in the line of the simulation models Statistical †, Markov‡, System Dynamics †, Discrete Event **†**, ABS is the most powerful one as it allows to incorporate / model the following:

- Linearity & Non-Linearity - the dynamics of the simulation can exhibit both linear and non-linear behaviour.
- Time - agents act over time, time is also the source of pro-activity.
- States - agents encapsulate some state which can be accessed and changed during the simulation.
- Feedback-Loops - because agents act continuously and their actions influence each other and themselves, feedback-loops are very common in ABS.
- Heterogeneity - although agents can have same properties like height, sex,... the value can be randomly distributed over the population, making it a heterogenous one. This is
- Interactions
- Spatiality & Networks

B. Functional Reactive Programming

So far we have considered only quite low-level approaches to structuring and composing functional programming: higher-order functions, laziness, monads and arrows. What we need is a programming paradigm built into pure functional programming which we can leverage to implement ABS. As already mentioned above, functional reactive programming (FRP) seems to be a highly promising approach. It is rather a lucky coincidence that Henrik Nilsson, one of the major contributor to the library Yampa, an implementation of FRP, is situated at the School of Computer Science of the University of Nottingham.

FRP is a paradigm for programming hybrid systems which combine continuous and discrete components. Time is explicitly modelled: there is a continuous and synchronous time flow. There have been many attempts to implement FRP in libraries which each has its benefits and deficits. The very first functional reactive language was Fran, a domain specific language for graphics and animation. At Yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa, the most recent FRP library [8]. The essence of FRP with Yampa is that one describes the system in terms of signal functions in a declarative manner using the EDSL of Yampa. During execution the top level signal functions will then be evaluated and return new signal functions which act as continuations. A major design goal for FRP is to free the programmer from 'presentation' details by providing the ability to think in terms of 'modeling'. It is common that an FRP program is concise enough to also serve as a specification for the problem it solves [9].

Yampa has been used in multiple agent-based applications: [10] uses Yampa for implementing a robot-simulation, [11] implement the classical Space Invaders game using Yampa, [12] implements a Pong-clone, the thesis of [13] shows how Yampa can be used for implementing a Game-Engine, [14] implemented a 3D first-person shooter game with the style of Quake 3 in Yampa. Note that although all these applications don't focus explicitly on agents all of them inherently deal with kinds of agents which share properties of classical agents: game-entities, robots,... Other fields in which Yampa was successfully used were programming of synthesizers, network routers, computer music development and has been successfully combined with monads [15].

This leads to the conclusion that Yampa is mature, stable and suitable to be used in functional ABS. This and the reason that we have the in-house knowledge lets us focus on Yampa. Also it is out-of-scope to do a in-depth comparison of the many existing FRP libraries.

FRP papers Functional Reactive Programming from First Principles (2000) - [9] Functional Reactive Programming Continued (2002) - [8] Arrows, Robots, and Functional Reactive Programming (2003) - [10] Functional Reactive Programming Refactored (2016) - [15]

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

```
type DTime = Double

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

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

Yampa provides a top-level function, running in the IO-Monad, which drives a signal-function by providing both input-values and time-deltas from callbacks. It is important to note that when visualizing a simulation one has in fact two flows of time: the one of the user-interface which always follows real-time flow, and the one of the simulation which could be sped up or slowed down. Thus it is important to note that if I/O of the user-interface (rendering, user-input) occurs within the simulations time-frame then the user-interfaces real-time flow becomes the limiting factor. Yampa provides the function embedSync which allows to embed a signal function within another one which is then run at a given ratio of the outer SF. This allows to give the simulation its own time-flow which is independent of the user-interface. We utilized this in the implementation of Recursive ABS (see Chapter **??**).

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

Yampa programming papers: The Yampa Arcade (2003) - [11] Functional Programming and 3D Games - [14] Game-Engine-Architectur (2010) - [13]

2) Arrowized Programming: Yampa makes heavy use of Arrows, a generalization of Monads, introduced by Hughes [16]. Arrows allow to parameterise over the input-type as well, which Monads do not allow: they only parameterise over the output-type. This allows to add static parts to an input of a function e.g. a time-delta. Also Arrows can be seen as processes TODO: explain. Both these properties of Arrows are the reason why Yampa is using Arrows (TODO: support by through the papers): it prevents time-leaks which occur when the time-delta parameter shows up visible to the function which then can be changed. Arrowized programming allows to hide exactly this time-delta parameter by passing it along in a point-free style and makes it thus impossible to mess around by the agent-implementer. Fortunately there has been developed a special notation, similar to the monad do-notation, for arrowized programming to increase readability [17]. The paper [18] gives a more in-depth explanation of Arrows, how to program with them and how to use the special arrow-notation. This being a paper for the ABS community, we won't go into depth about arrowized programming as it would be out of the scope of this paper but the concept should become clear from the code-examples in the later sections.

C. NetLogo

One can look at NetLogo as a functional approach to ABMS which comes with its own EDSL. Our approach differs fundamentally in the following way - untyped - no side-effects possible - no direct access to other agents, communication happens through asynchronous messages or synchronized conversations - powerful time-semantics which NetLogo completely lacks

D. Examples

1) SIRS: [ ] SIRS: timed transitions using after, occasionally sending messages, transitions on messages
$\frac{\mathrm{d}S}{\mathrm{d}t} = -infectionRate$
$\frac{\mathrm{d}I}{\mathrm{d}t} = infectionRate - recoveryRate$
$\frac{\mathrm{d}R}{\mathrm{d}t} = recoveryRate$

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

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

2) Wildfire: [ ] WildFire: rate transitions, occasionally sending messages, transition on messages

III. The formal Agent-Model

Functional programming in Haskell with its strong static type-system can be seen as to be much more formal than object-oriented programming in Java and C++. Where in the latter one e.g. a Class-Diagram in UML would be created, in our functional approach we create a formal model of our agent which can then be easily translated to Haskell.

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

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

A. Id

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

B. State

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

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

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

C. Messages

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

```
data SIRSMsg = Contact SIRSState
```

D. Environment-Cell

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

E. Behaviour

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

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

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

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

IV. Environment

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

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

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

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

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

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

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

V. Functional Reactive ABS

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

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

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

A. Messaging

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

B. Conversations

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

C. Iteration-Strategies

Building on the foundations laid out in my paper about iteration-strategies in Appendix **??**, we implement two of the four strategies: sequential- and parallel-strategy. We deliberately ignore the concurrent- and actor-strategy for now and leave this for further research [1]. Implementing iteration-strategies using Haskell and FRP is not as straight-forward as in e.g. Java because one does not have mutable data which can be updated in-place. Although my work on programming

[1] Also both strategies would require running in the STM-Monad, which is not possible with Yampa. The work of Ivan Perez in [15] implemented a library called Dunai, which is the same as Yampa but capable of running in an arbitrary Monad.

paradigms in Appendix **??** did not take FRP into account, general concepts apply equally as well.

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

2) Parallel: The parallel strategy is *much* easier to implement than the sequential but is of course not applicable to all models because of it different semantics. Basically this strategy is implemented as a map over all agents which calls each agent-behaviour signal-function with the agents AgentIn to retrieve the new AgentOut. Then the messages are distributed amongst all agents. A problem in this strategy is that the environment is duplicated to each agent and then each agent can work on it and return a changed environment. Thus after one iteration there are n versions of environments where n is equals to the number of agents. These environments must then be collapsed into a final one which is always domain-specific thus needs to be done through a function provided in the environment itself.

D. Environment

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

Each agent has a copy of the environment passed in through the AgentIn and can change it by passing a changed version of the environment out through AgentOut. In the sequential update-strategy the environment of the agent i will then be passed to all agents i + 1 AgentIn in the current iteration and override their old environment. Thus all steps of changes made to the environment are visible in the AgentOuts. The last environment is then the final environment int he current iteration and will be returned by the callback function together with the current AgentOuts. In the parallel update-strategy the environment is duplicated to each agent and then each agent can work on it and return the changed environment. Thus after one iteration there are n versions of environments where n is equals to the number of agents. These environments must then be collapsed into a final one which is always domain-specific thus needs to be done through a function provided in the environment itself. In both the sequential and parallel update-strategy after one iteration there is one single environment left. An environment can have an optional behaviour which allows the environment to update its cells. This is a regular SF thus having also the time of the simulation available. Note that the environment cannot send messages to agents because it is not an agent itself. An example of an environment behaviour would be to regrow some good on each cell according to some rate per time-unit (inspired by SugarScape regrowing of Sugar).

E. Time-Semantics

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

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

F. Agents as Signals

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

G. Time-Sampling

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

H. System Dynamics

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

I. Discrete Event Simulation

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

VI. Examples

A. Agent-Based SIR

Advantage: can incorporate networking or spatial effects. In this example it runs within a discrete 2D environment. FrABMS library also has the ability for networks.

Hypothesis: high-frequency sampling is required when there are rates involved e.g. occacionally. TODO: compare fully-connected SIR to System Dynamics solution TODO: look into sequential (shuffled/nonshuffled) vs parallel

B. System Dynamics SIR

Because we have the powerful time-semantic features of yampa at hand which allows to sample a system at very high frequency with continuous time at hand we can easily implement System Dynamics. Using parallel iteration-strategy (no shuffling of agents) Thus in a way we can see FrABMS to be a hybrid approach between ABMS and System Dynamics.

1) Stocks: are completely defined by the formula Initial-Value + Integrate (0 to t) by dt (inflow - outflow)

by messages stocks communicate their current value to the flows which require them. receive by messages the current vale of all flows relevant to them

2) Flows: Are stateless and can calculate any rate

receive through messages the current values of their relevant Stocks send their current flow-value to the relevant Stocks

VII. Discussion

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

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

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

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

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

VIII. Testing

[19]

A. Time-Traveling

[20]

IX. Conclusion and future research

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

In his 1st year report about Functional Reactive GUI programming, Ivan Perez [2] writes: "FRP tries to shift the direction of data-flow, from message passing onto data dependency. This helps reason about what things are over time, as opposed to how changes propagate". This of course raises the question whether FRP is *really* the right approach, because the way we implement ABS, message-passing is an essential concept. It is important to emphasis that agent-relations in interactions are never fixed in advance and are completely dynamic, forming a network. Maybe one has to look at message passing in a different way in FRP, and to view and model it as a data-dependency but it is not clear how this can be done. The question is whether there is a mechanism in which we have explicit data-dependency but which is dynamic like message-passing but does not try to fake method-calls? Maybe the concept of conversations (see above) are a way to go but we leave this for further research at the moment.

future research: STM in FrABS: run them in parallel but concurrently and advance time through a separate STM loop in the main loop. constant time should then keep the agents constant unless some discrete event happens e.g. message arrives. sugarscape wouldnt work but FrSIR. but still we have

[2]main author of the paper [15]

random results. also i could implement this in java unless i use STM specific stuff

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

Acknowledgments

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

References

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

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

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

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

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

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

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

[8] 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

[9] Z. Wan and P. Hudak, "Functional Reactive Programming from First Principles," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: ACM, 2000, pp. 242–252. [Online]. Available: http://doi.acm.org/10.1145/349299.349331

[10] 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

[11] 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

[12] H. Nilsson and I. Perez, "Declarative Game Programming: Distilled Tutorial," in *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '14. New York, NY, USA: ACM, 2014, pp. 159–160. [Online]. Available: http://doi.acm.org/10.1145/2643135.2643160

[13] G. Meisinger, "Game-Engine-Architektur mit funktional-reaktiver Programmierung in Haskell/Yampa," Master, Fachhochschule Obersterreich - Fakultt fr Informatik, Kommunikation und Medien (Campus Hagenberg), Austria, 2010.

[14] C. Mun Hon, "Functional Programming and 3d Games," Ph.D. dissertation, University of New South Wales, Sydney, Australia, 2005.

[15] 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

[16] J. Hughes, "Generalising Monads to Arrows," *Sci. Comput. Program.*, vol. 37, no. 1-3, pp. 67–111, May 2000. [Online]. Available: http://dx.doi.org/10.1016/S0167-6423(99)00023-4

[17] 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

[18] J. Hughes, "Programming with Arrows," in *Proceedings of the 5th International Conference on Advanced Functional Programming*, ser. AFP'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 73–129. [Online]. Available: http://dx.doi.org/10.1007/11546382_2

[19] 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

[20] I. Perez, "Back to the Future: Time Travel in FRP," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2017. New York, NY, USA: ACM, 2017, pp. 105–116. [Online]. Available: http://doi.acm.org/10.1145/3122955.3122957

[21] J. M. Epstein, *Agent_Zero: Toward Neurocognitive Foundations for Generative Social Science*. Princeton University Press, Feb. 2014, google-Books-ID: VJEpAgAAQBAJ.

[22] T. Schelling, "Dynamic models of segregation," *Journal of Mathematical Sociology*, vol. 1, 1971.

[23] M. A. Nowak and R. M. May, "Evolutionary games and spatial chaos," *Nature*, vol. 359, no. 6398, pp. 826–829, Oct. 1992. [Online]. Available: http://www.nature.com/nature/journal/v359/n6398/abs/359826a0.html

[24] B. A. Huberman and N. S. Glance, "Evolutionary games and computer simulations," *Proceedings of the National Academy of Sciences*, vol. 90, no. 16, pp. 7716–7718, Aug. 1993. [Online]. Available: http://www.pnas.org/content/90/16/7716

[25] U. Wilensky and W. Rand, *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo*. MIT Press, 2015. [Online]. Available: https://www.amazon.co.uk/Introduction-Agent-Based-Modeling-Natural-Engineered/dp/0262731894

[26] T. Breuer, M. Jandaka, M. Summer, and H.-J. Vollbrecht, "Endogenous leverage and asset pricing in double auctions," *Journal of Economic Dynamics and Control*, vol. 53, no. C, pp. 144–160, 2015. [Online]. Available: http://econpapers.repec.org/article/eeedyncon/v_3a53_3ay_3a2015_3ai_3ac_3ap_3a144-160.htm

[27] 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

[28] N. Bostrom, "Are We Living in a Computer Simulation?" *The Philosophical Quarterly*, vol. 53, no. 211, pp. 243–255, 2003. [Online]. Available: http://dx.doi.org/10.1111/1467-9213.00309

[29] E. Steinhart, "Theological Implications of the Simulation Argument," *Ars Disputandi: The Online Journal for Philosophy of Religion*, vol. 10, pp. 23–37, 2010.

APPENDIX A
EXAMPLES

In this appendix we give a list of all the examples we have implemented and discuss implementation details relevant [3]. The examples were implemented as use-cases to drive the development of *FrABS* and to give code samples of known models which show how to use this new approach. Note that we do not give an explanation of each model as this would be out of scope of this paper but instead give the major references from which an understanding of the model can be obtained.

We distinguish between the following attributes

- Implementation - Which style was used? Either Pure, Monadic or Reactive. Examples could have been implemented in all of them.
- Yampa Time-Semantics - Does the implemented model make use of Yampas time-semantics e.g. occasional, after,...? Yes / No.
- Update-Strategy - Which update-strategy is required for the given example? It is either Sequential or Parallel or both. In the case of Sequential Agents may be shuffled or not.
- Environment - Which kind of environment is used in the given example? Possibilities are 2D/3D Discrete/-Continuous or Network. In case of a Parallel Update-Strategy, collapsing may become necessary, depending on the semantics of the model. Also it is noted if the environment has behaviour. Note that an implementation may also have no environment which is noted as None. Although every model implemented in *FrABs* needs to set up some environment, it is not required to use it in the implementation.
- Recursive - Is this implementation making use of the recursive features of *FrABS* Yes/No (only available in sequential updating)?
- Conversations - Is this implementation making use of the conversations features of *FrABS* Yes/No (only available in sequential updating)?

A. Sugarscape

This is a full implementation of the famous Sugarscape model as described by Epstein & Axtell in their book [1]. The model description itself has no real time-semantics, the agents act in every time-step. Only the environment may change its behaviour after a given number of steps but this is easily expressed without time-semantics as described in the model by Epstein & Axtell [4].

[3]The examples are freely available under https://github.com/thalerjonathan/phd/tree/master/coding/libraries/frABS/examples

[4]Note that this implementation has about 2600 lines of code which - although it includes both a pure and monadic implementation - is significant lower than e.g. the Java-implementation http://sugarscape.sourceforge.net/ with about 6000. Of course it is difficult to compare such measures as we do not include FrABS itself into our measure.

Implementation	Pure, Monadic
Yampa Time-Semantics	No
Update-Strategy	Sequential, shuffling
Environment	2D Discrete, behaviour
Recursive	No
Conversations	Yes

B. Agent_Zero

This is an implementation of the *Parable 1* from the book of Epstein [21].

Implementation	Pure, Monadic
Yampa Time-Semantics	No
Update-Strategy	Parallel, Sequential, shuffling
Environment	2D Discrete, behaviour, collapsing
Recursive	No
Conversations	No

C. Schelling Segregation

This is an implementation of [22] with extended agent-behaviour which allows to study dynamics of different optimization behaviour: local or global, nearest/random, increasing/binary/future. This is also the only 'real' model in which the recursive features were applied [5].

Implementation	Pure
Yampa Time-Semantics	No
Update-Strategy	Sequential, shuffling
Environment	2D Discrete
Recursive	Yes (optional)
Conversations	No

D. Prisoners Dilemma

This is an implementation of the Prisoners Dilemma on a 2D Grid as discussed in the papers of [23], [24] and TODO: cite my own paper on update-strategies.

TODO: implement

E. Heroes & Cowards

This is an implementation of the Heroes & Cowards Game as introduced in [25] and discussed more in depth in TODO: cite my own paper on update-strategies.

TODO: implement

F. SIRS

This is an early, non-reactive implementation of a spatial version of the SIRS compartment model found in epidemiology. Note that although the SIRS model itself includes time-semantics, in this implementation no use of Yampas facilities were made. Timed transitions and making contact was implemented directly into the model which results in contacts being made on every iteration, independent of the sampling time. Also in this sample only the infected agents make contact with others, which is not quite correct when wanting to approximate the System Dynamics model (see

[5]The example of Recursive ABS is just a plain how-to example without any real deeper implications.

below). It is primarily included as a comparison to the later implementations (Fr*SIRS) of the same model which make full use of *FrABS* and to see the huge differences the usage of Yampas time-semantics can make.

Implementation	Pure, Monadic
Yampa Time-Semantics	No
Update-Strategy	Parallel, Sequential with shuffling
Environment	2D Discrete
Recursive	No
Conversations	No

G. Reactive SIRS

This is the reactive implementations of both 2D spatial and network (complete graph, Erdos-Renyi and Barbasi-Albert) versions of the SIRS compartment model. Unlike SIRS these examples make full use of the time-semantics provided by Yampa and show the real strength provided by *FrABS*.

Implementation	Reactive
Yampa Time-Semantics	Yes
Update-Strategy	Parallel
Environment	2D Discrete, Network
Recursive	No
Conversations	No

H. System Dynamics SIR

This is an emulation of the System Dynamics model of the SIR compartment model in epidemiology. It was implemented as a proof-of-concept to show that *FrABS* is able to implement even System Dynamic models because of its continuous-time and time-semantic features. Connections between stocks & flows are hardcoded, after all System Dynamics completely lacks the concept of spatial- or network-effects. Note that describing the implementation as Reactive may seem not appropriate as in System Dynamics we are not dealing with any events or reactions to it - it is all about a continuous flow between stocks. In this case we wanted to express with Reactive that it is implemented using the Arrowized notion of Yampa which is required when one wants to use Yampas time-semantics anyway.

Implementation	Reactive
Yampa Time-Semantics	Yes
Update-Strategy	Parallel
Environment	None
Recursive	No
Conversations	No

I. WildFire

This is an implementation of a very simple Wildfire model inspired by an example from AnyLogic™ with the same name.

Implementation	Reactive
Yampa Time-Semantics	Yes
Update-Strategy	Parallel
Environment	2D Discrete
Recursive	No
Conversations	No

J. Double Auction

This is a basic implementation of a double-auction process of a model described by [26]. This model is not relying on any environment at the moment but could make use of networks in the future for matching offers.

Implementation	Pure, Monadic
Yampa Time-Semantics	No
Update-Strategy	Parallel
Environment	None
Recursive	No
Conversations	No

K. Policy Effects

This is an implementation of a model inspired by Uri Wilensky [6]: "Imagine a room full of 100 people with 100 dollars each. With every tick of the clock, every person with money gives a dollar to one randomly chosen other person. After some time progresses, how will the money be distributed?"

Implementation	Monadic
Yampa Time-Semantics	No
Update-Strategy	Parallel
Environment	Network
Recursive	No
Conversations	No

L. Proof of concepts

1) Recursive ABS: This example shows the very basics of how to implement a recursive ABS using *FrABS*. Note that recursive features only work within the sequential strategy.

Implementation	Pure
Yampa Time-Semantics	No
Update-Strategy	Sequential
Environment	None
Recursive	Yes
Conversations	No

2) Conversation: This example shows the very basics of how to implement conversations in *FrABS*. Note that conversations only work within the sequential strategy.

Implementation	Pure
Yampa Time-Semantics	No
Update-Strategy	Sequential
Environment	None
Recursive	No
Conversations	Yes

[6]http://www.decisionsciencenews.com/2017/06/19/counterintuitive-problem-everyone-room-keeps-giving-dollars-random-others-youll-never-g

APPENDIX B
RECURSIVE AGENT-BASED SIMULATION

The idea for this paper arose from my idea of *anticipating agents*, which can project their actions in the future. Because this paper is not as polished as the draft for programming paradigms, we opted not to include it as an appendix and only give its basic ideas and results for the experiments conducted so far. Note that we were not able to find any research regarding recursive ABS [7]. In Recursive ABS agents are able to halt time and 'play through' an arbitrary number of actions, compare their outcome and then to resume time and continue with a specifically chosen action e.g. the best performing or the one in which they haven't died. More precisely, what we want is to give an agent the ability to run the simulation recursively a number of times where the this number is not determined initially but can depend on the outcome of the recursive simulation. So Recursive ABS gives each Agent the ability to run the simulation locally from its point of view to anticipate its actions in the future and change them in the present. We investigate the famous Schelling Segregation [22] and endow our agents with the ability to project their actions into the future by recursively running simulations. Based on the outcome of the recursions they are then able to determine whether their move increases their utility in the future or not. The main finding for now is that it does not increase the convergence speed to equilibrium but can lead to extreme volatility of dynamics although the system seems to be near to complete equilibrium. In the case of a 10x10 field it was observed that although the system was nearly in its steady state - all but one agent were satisfied - the move of a single agent caused the system to become completely unstable and depart from its near-equilibrium state to a highly volatile and unstable state.

This approach of course rises a few questions and issues. The main problem of our approach is that, depending on ones view-point, it is violating the principles of locality of information and limit of computing power. To recursively run the simulation the agent which initiates the recursion is feeding in all the states of the other agents and calculates the outcome of potentially multiple of its own steps, each potentially multiple recursion-layers deep and each recursion-layer multiple time-steps long. Both requires that each agent has perfect information about the complete simulation *and* can compute these 3-dimensional recursions, which scale exponentially. In the social sciences where agents are often designed to have only very local information and perform low-cost computations it is very difficult or impossible to motivate the usage of recursive simulations - it simply does not match the assumptions of the real world, the social sciences want to model. In general simulations, where it is much more commonly accepted to assume perfect information and potentially infinite amount of computing power this approach is easily motivated by a constructive argument: it is possible to build, thus we build it. Another fundamental question regards the meaning and epistemology behind an entity running simulations. Of course, this strongly depends on the context: in ACE it may be understood as a search for optimizing behaviour, in Social Simulation it may be interpreted as a kind of free will: the agent who is initiating the recursion can be seen as 'knowing' that it is running inside a simulation, thus in this context free will is seen as being able to anticipate ones actions and change them. When talking about recursion it is always the question of the depth of the recursion and because as we are running on computers we need to terminate at some point. Accelerating Turing machines (also known as Zeno Machine) are theoretically able to calculate an infinite regress but this raises again epistemological questions and can be seen as having religious character as discussed e.g. in Tiplers Omega Point, Bostroms simulation argument [28] and its theological implications [29]. So the ultimate question this research leaves is what the outcome would be when running a recursive ABS on a Zeno Machine/Accelerated Turing Machine? [8]

At the moment this idea lies dormant as the intention was just to develop it far enough to give a proof-of-concept and see some results. Having achieved this we arrived at the conclusion, that the results are not really ground-breaking. This stems from the fact that Schelling segregation is not the best model to demonstrate this technique and that we are thus lacking the right model in which recursive ABS is the real killer-feature. Also to pursue this direction further and treat it in-depth, would require much more time and give the PhD a complete different spin. Still it is useful in supporting our move towards pure functional ABS as we are convinced that recursion is comparably easy to implement because the language is built on it and due to the lack of side-effects [9].

[7] We found a paper on recursive simulation in general [27] which focuses on military simulation implemented in C++. Its main findings are that deterministic models seem to benefit significantly from using recursions of the simulation for the decision making process and that when using stochastic models this benefit seems to be lost.

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

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