Towards pure functional agent-based simulation

Jonathan Thaler
School of Computer Science
University of Nottingham
jonathan.thaler@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 seminal Sugarscape model in Haskell in our 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 ABS as it allows to incorporate discrete time-semantics similar to Discrete Event Simulation and continuous time-flows as in System Dynamics. In this paper we will show the novel approach of functional reactive ABS through the example of the SIR model, discuss implications, benefits and best practices.

Index Terms

Haskell, Functional Programming, Verification

I. Introduction

In this paper we investigate how agent-based simulation can be approach from a pure functional direction using the programming language Haskell. So far no in-depth research has been conducted on this subject because so far agent-based simulation was always thought of as being best implemented in object-oriented languages like C++, Java and Python. As will become apparent throughout this paper, a pure functional approach needs to approach various concepts of ABS very different which leads to a very different approach to ABS but makes concepts which where blurred and implicit very explicit and clear. This papers major contribution is that it reveals these implicit concept and provides a formal view on ABS and a pure functional implementation of these concepts.

II. Agent-Based Simulation Defined

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. Epstein [1] 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 methods *Statistic* †, *Markov* ‡, *System Dynamics* §, *Discrete Event* $^{\mp}$, ABS is the most recent development and the most powerful one as it subsumes it's predecessors features and goes beyond:

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

A. Deriving central concepts

Before we can approach a functional view on ABS, we need to identify the central concepts of ABS on a more technical level. Unfortunately there does not exist a commonly agreed technical definition of ABS but we can draw inspiration from the closely related field of Multi-Agent Systems (MAS). It is important to understand that MAS and ABS are two different fields where in MAS the focus is much more on technical details implementing a system of interacting intelligent agents within a highly complex environment with the focus primarily on solving AI problems.

1) Agents: The central aspect of ABS is the concept of an agent. In MAS [2], [3] agents can be informally defined as:

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

2) Environment: The other important concept is the one of an environment. In MAS [2], [3] one distinguishes between different types of environments (based on [4]):

- Accessible vs. inaccessible - in an accessible environment an agent can obtain complete and accurate information from the environment. In ABS environments are generally implemented as being accessible.
- Deterministic vs. non-deterministic - in a deterministic environment the actions of an agent have no uncertainty and are guaranteed to have a single effect. In ABS environments are generally implemented as being deterministic.
- Static vs. dynamic - a static environment only changes due to the agents actions whereas a dynamic one has other processes which operate on it. In ABS both static and dynamic environments are common.
- Discrete vs. continuous - a discrete environment has only a fixed, finite number of states and actions whereas a continuous is potentially unlimited. In ABS both discrete and continuous environments are common.

Note that in MAS the focus is much more on the environment rather than on the agents where the environment is almost always a highly complex one and the agents may intelligently act on it. In ABS the focus is rather on the agents and their interactions where the environment plays a role but is not of central interest as it is almost always deterministic.

B. Deriving a formal view

In order to explore how we can implement an ABS in a pure functional way we need a sufficiently formal view on it. This will help us expressing the concepts in Haskell as formal, mathematical specifications translate easily into functional programming. There exists formalisations of MAS [2] but unfortunately they are not very helpful in our context as its formalization is tailored much more towards optimizing, intelligent and reasoning behaviour of agents within a highly complex and uncertain environment. What we need for ABS is a more agent-oriented approach:

1) An ABS is a simulation over time in which time is advanced either in discrete or continuous time-steps where discrete means advancing by a natural number time-delta and continuous by a real-valued time-delta. So we have a potentially infinite stream of time-steps starting at t=0 advancing by some fixed time-delta.
2) At each time-step all agents are allowed to act which is the source of their proactivity because it allows them to initiate actions on their own. Of course such actions are always time-dependent - be it explicitly like executing actions *after* a specific time, or be it implicit like executing actions every time-delta - but this is the only way of implementing proactivity in a computer system.
3) In each step an agent should be able to read/write the environment. TODO: orderings? when are changes visible?
4) In each step an agent should be able to interact with other agents through communication.
5) In each step an agent should be able to update its internal state.
6) Depending on its type, the environment must also be allowed to act in each time-step.
7) In general we can thus see an agent to exhibit both time-dependent and reactive behaviour: it can act continuously or discretely, depending on how the time is advanced and exhibit reactive behaviour which means it can react to changing environment or agents.
8) The interactions between agents their update-state and environment forms a feedback as the state of time ti forms the input state on which to act at time-step ti+1

III. A FUNCTIONAL VIEW

Due to the fundamentally different approaches of pure Functional Programming (pure FP) an ABS needs to be implemented fundamentally different as well compared to traditional object-oriented approaches (OO). We face the following challenges:

1) How can we represent an Agent?
   In OO the obvious approach is to map an agent directly onto an object which encapsulates data and provides methods which implement the agents actions. Obviously we don't have objects in pure FP thus we need to find a different approach to represent the agents actions and to encapsulate its state.
2) How can we represent state in an Agent?
   In the classic OO approach one represents the state of an Agent explicitly in mutable member variables of the object which implements the Agent. As already mentioned we don't have objects in pure FP and state is immutable which leaves us with the very tricky question how to represent state of an Agent which can be actually updated.

3) How can we implement proactivity of an Agent?
   In the classic OO approach one would either expose the current time-delta in a mutable variable and implement time-dependent functions or ignore it at all and assume agents act on every step. At first this seems to be not a big deal in pure FP but when considering that it is yet unclear how to represent Agents and their state, which is directly related to time-dependent and reactive behaviour it raises the question how we can implement time-varying and reactive behaviour in a purely functional way.
4) How can we implement the agent-agent interaction?
   In the classic OO approach Agents can directly invoke other Agents methods which makes direct Agent interaction *very* easy. Again this is obviously not possible in pure FP as we don't have objects with methods and mutable state inside.
5) How can we represent an environment and its various types?
   In the classic OO approach an environment is almost always a mutable object which can be easily made dynamic by implementing a method which changes its state and then calling it every step as well. In pure FP we struggle with this for the same reasons we face when deciding how to represent an Agent, its state and proactivity.
6) How can we implement the agent-environment interaction?
   In the classic OO approach agents simply have access to the environment either through global mechanisms (e.g. Singleton or simply global variable) or passed as parameter to a method and call methods which change the environment. Again we don't have this in pure FP as we don't have objects and globally mutable state.
7) How can we step the simulation?
   In the classic OO approach agents are run one after another (with being optionally shuffled before to uniformly distribute the ordering) which ensures mutual exclusive access in the agent-agent and agent-environment interactions. Obviously in pure FP we cannot iteratively mutate a global state.

A. *Agent representation, state and proactivity*

Whereas in imperative programming (the OO which we refer to in this paper is built on the imperative paradigm) the fundamental building block is the destructive assignment, in FP the building blocks are obviously functions which can be evaluated. Thus we have no other choice than to represent our Agents using a function which implements their behaviour. This function must be time-aware somehow and allow us to react to time-changes and inputs. Fortunately there exists already an approach to time-aware, reactive programming which is termed Functional Reactive Programming (FRP). This paradigm has evolved over the year and current modern FRP is built around the concept of a signal-function which transforms an input-signal into an output-signal. An input-signal can be seen as a time-varying value. Signal-functions are implemented as continuations which allows to capture local state using closures. Modern FRP also provides feedback functions which provides convenient methods to capture and update local state from the previous time-step with an initial state provided at time = 0.

- time is represented using the FRP concept: Signal-Functions which are sampled at (fixed) time-deltas, the dt is never visible directly but only reflected in the code and read-only. - no method calls =¿ continuous data-flow instead

Viewing agent-agent interaction as simple method calls implies the following: - it takes no time - it has a synchronous and transactional character - an agent gives up control over its data / actions or at least there is always the danger that it exposes too much of its interface and implementation details. - agents equals objects, which is definitely NOT true. Agents

data-flow synchronous agent transactions

- still need transactions between two agents e.g. trading occurs over multiple steps (makeoffer, accept/refuse, finalize/abort) -¿ exactly define what TX means in ABS -¿ exclusive between 2 agents -¿ state-changes which occur over multiple steps and are only visible to the other agents after the TX has commited -¿ no read/write access to this state is allowed to other agents while the TX is active -¿ a TX executes in a single time-step and can have an arbitrary number of tx-steps -¿ it is easily possible using method-calls in OOP but in our pure functional approach it is not possible -¿ parallel execution is being a problem here as TX between agents are very easy with sequential -¿ an agent must be able to transact with as many other agents as it wants to in the same time-step -¿ no time passes between transactions =¿ what we need is a 'all agents transact at the same time' -¿ basically we can implement it by running the SFs of the agents involved in the TX repeatedly with dt=0 until there are no more active TXs -¿ continuations (SFs) are perfectly suited for this as we can 'rollback' easily by using the SF before the TX has started

B. *Environment representation and interaction*

pro-active vs. inactive read-only vs. read/write

no global shared mutable environment, having different options: - non-active read-only (SIR): no agent, as additional argument to each agent - pro-active read-only (?): environment as agent, broadcast environment updates as data-flow - non-active read/write (?): no agent, shared data as STM as additional argument to each agent - pro-active read/write (Sugarscape): environment as, shared data as STM as additional argument to each agent

C. *Stepping the simulation*

- parallel update only, sequential is deliberately abandoned due to: -¿ reality does not behave this way -¿ if we need transactional behaviour, can use STM which is more explicit -¿ it is translates directly to a map which is very easy to reason about (sequential is basically a fold which is much more difficult to reason about) -¿ is more natural in functional programming -¿ it exists for 'transactional' reasons where we need mutual exclusive access to environment / other agents -¿ we provide a more explicit mechanism for this: Agent Transactions

IV. Examples

TODO: the main punch is that our approach combines the best of the three simulation methodologies: - SD part: it can represent continuous time (as well as discrete) with continuous data-flows from agents which act at the same time (parallel update), can express the formulas directly in code, there exists also a small EDSL for expressing SD in our approach, can guarantee reproducibility and no drawing of random-numbers in our approach -¿ drawback over real SD: none known so far - DES part: it can represent discrete time with events occurring at discrete points in time which cause an instant change in the system -¿ drawback over real DES: time does not advance discretely to the next event which results of course not in the performance of a real DES system - ABS part: the entities of the system (=agents) can be heterogenous and pro-active in time and can have arbitrary neighbourhood (2d/3d discrete/continuous, network,...) -¿ drawback over classic ABS: none known so far

TODO: give examples of all 3 approaches: SD & ABS: SIR model, DES: simulation of a queuing system

A. *Library*

In the course of the research for this paper we implemented the library *Chimera* [1] in Haskell which implements all the concepts introduced in this paper and allows to do ABS in a pure functional way in Haskell. As use-cases to drive the research and test our concepts we implemented a number of more or less well known examples from ABS, see Appendix A for a list of all the examples and the concepts used.

V. Discussion

Our purely functional approach has a number of fundamental implications which change the way one has to think about agents and ABS in general as it makes a few concepts which were so far hidden or implicitly assumed, now explicit.

A. *Agents as Signals*

Our approach of using signal-functions has the direct implication that we can view and implement agents as time-dependent signals. A time-dependent function should stay constant when time does not advance (when the system is stepped with time-delta of 0). For this to happen with our agents requires them to act only on time-dependent functions. TODO: further discussion using my work on agents as signals in the first draft on FrABS.

TODO: it doesn't make sense for an agent to act 'always', an agents behaviour needs to have some time-dependent parameter e.g. doEvery 1.0. If this is omitted then one makes one dependent directly on the Time-Delta.

B. *System Dynamics*

Due to the parallel execution of the agents signal-functions, the ability to iterate the simulation with continuous time, the notion of continuous data-flow between agents and compile time guarantees of absence of non-deterministic side-effects and random-number generators allows us to directly express System Dynamic models. Each stock and flow becomes an agent and are connected using data-flows using hard-coded agent ids. The integrals over time which occur in a SD model are directly translated to pure functional code using the *integral* primitive of FRP - our implementation is then correct by definition. See Appendix TODO for an example which implements the SIR model (TODO: cite mckendrick) in SD using our continuous ABS approach.

VI. Conclusions and future work

A. *Advantages*

- being explicit and polymorph about side-effects: can have 'pure' (no side-effects except state), 'random' (can draw random-numbers), 'IO' (all bets are off), STM (concurrency) agents

- hybrid between SD and ABS due to continuous time AND parallel dataFlow (parallel update-strategy)

- being update-strategy polymorph (TODO: this is just an asumption atm, need to prove this): 4 different update-strategies, one agent implementation

- parallel update-strategy: lack of implicit side-effects makes it work without any danger of data-interference

[1]The library is freely available on GitHub: https://github.com/thalerjonathan/chimera. It is planned that it will be put on Hackage in the future.

- recursive simulation - reasoning about correctness - reasoning about dynamics - testing with quickcheck much more convenient - expressivity: -¿ 1:1 mapping of SD to code: can express the SD formulas directly in code -¿ directly expressing state-charts in code

B. 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.

C. Future Work

TODO: what we need to show / future work - can we do DES? e.g. single queue with multiple servers? also specialist vs. generalist - reasoning about correctness: implement Gintis & Ionescous papers - reasoning about dynamics: implement Gintis & Ionescous papers

Acknowledgments

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

References

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

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

[3] G. Weiss, *Multiagent Systems*. MIT Press, Mar. 2013, google-Books-ID: WY36AQAAQBAJ.

[4] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

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

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

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

[8] 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

[9] 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

[10] 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

[11] 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

Appendix A
Examples

In this appendix we give a list of all the examples we have implemented and discuss implementation details relevant [2]. 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 [5]. 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 [3].

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 [6].

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 [7] 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 [4].

[2]The examples are freely available under https://github.com/thalerjonathan/chimera/tree/master/examples

[3]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.

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

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 [8], [9] 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 [10] 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 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 [11]. 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 [5]: ”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

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

Appendix B
Functional-Reactive code of the System Dynamics SIR model

```haskell
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
```