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

Agent-Based Simulation, Functional Programming, Functional Reactive Programming, Haskell

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. Background

In this background section we will introduce the concepts behind functional programming and functional reactive programming as they form the very foundation of our approach to pure functional Agent-Based Simulation. Note that both fields build on each other and are vast and complex. A more in-depth handling of both subjects are out of scope of this paper and we refer the reader to additional resources where appropriate. In this paper we try to avoid too much technical details and present only the fundamental concepts behind our approach. [1].

A. Functional Programming

In his 1977 ACM Turing Award Lecture, John Backus [2] fundamentally critizied imperative programming for its deep flaws and proposed a functional style of programming to overcome the limitations of imperative programming [1]. The main criticism is its use of *state-transition with complex states* and the inherent semantics of state-manipulation. In the end an imperative program consists of a number of assign-statements resulting in side-effects on global mutable state which makes reasoning about programs nearly impossible. Backus proposes the so called *applicative* computing, which he termes *functional programming* which has its foundations in the Lambda Calculus [2]. The main idea behind it is that programming follows a declarative rather than an imperative style of programming: instead of describing *how* something is computed, one describes *what* is computed. This concept abandons variables, side-effects and (global) mutable state and resorts to the simple core of function application, variable substitution and binding of the Lambda Calculus. Although possible and an important step to understand the very foundations, one does not do functional programming in the Lambda Calculus [3], as one does not do imperative programming in a Turing Machine.

MacLennan [4] defines Functional Programming as a methodology and identifies it with the following properties (amongst others):

1) It is programming without the assignment-operator.

[1]For interested readers, we refer to our companion paper TODO: cite / refer to existing repository. In it we introduce the functional programming community to ABS by deriving our approach step-by-step by implementing an agent-based implementation of the SIR model. In this companion paper we assume good knowledge and beyond of the fundamental concepts of FP and FRP and provide in-depth discussions of technical details.

[2]One of the giants of Computer Science, a main contributor to Fortran - an imperative programming language.

2) It allows for higher levels of abstraction.
3) It allows to develop executable specifications and prototype implementations.
4) It is connected to computer science theory.
5) Suitable for Parallel Programming.
6) Algebraic reasoning.

[5] defines Functional Programming as ”a computer programming paradigm that relies on functions modelled on mathematical functions.” Further they explicate that it is

- in Functional programming programs are combinations of expressions
- Functions are *first-class* which means the can be treated like values, passed as arguments and returned from functions.

[4] makes the subtle distinction between *applicative* and *functional* programming. Applicative programming can be understood as applying values to functions where one deals with pure expressions:

- Value is independent of the evaluation order.
- Expressions can be evaluated in parallel.
- Referential transparency.
- No side effects.
- Inputs to an operation are obvious from the written form.
- Effects to an operation are obvious from the written form.

Note that applicative programming is not necessarily unique to the functional programming paradigm but can be emulated in an imperative language e.g. C as well. Functional programming is then defined by [4] as applicative programming with *higher-order* functions. These are functions which operate themselves on functions: they can take functions as arguments, construct new functions and return them as values. This is in stark contrast to the *first-order* functions as used in applicative or imperative programming which just operate on data alone. Higher-order functions allow to capture frequently recurring patterns in functional programming in the same way like imperative languages captured patterns like GOTO, while-do, if-then-else, for. Common patterns in functional programming are the map, fold, zip, operators. So functional programming is not really possible in this way in classic imperative languages e.g. C as you cannot construct new functions and return them as results from functions [3].

The equivalence in functional programming to to the *;* operator of imperative programming which allows to compose imperative statements is function composition. Function composition has no side-effects as opposed to the imperative ; operator which simply composes destructive assignment statements which are executed after another resulting in side-effects. At the heart of modern functional programming is monadic programming which is polymorphic function composition: one can implement a user-defined function composition by allowing to run some code in-between function composition - this code of course depends on the type of the Monad one runs in. This allows to emulate all kind of effectful programming in an imperative style within a pure functional language. Although it might seem strange wanting to have imperative style in a pure functional language, some problems are inherently imperative in the way that computations need to be executed in a given sequence with some effects. Also a pure functional language needs to have some way to deal with effects otherwise it would never be able to interact with the outside-world and would be practically useless. The real benefit of monadic programming is that it is explicit about side-effects and allows only effects which are fixed by the type of the monad - the side-effects which are possible are determined statically during compile-time by the type-system. Some general patterns can be extracted e.g. a map, zip, fold over monads which results in polymorphic behaviour - this is the meaning when one says that a language is polymorphic in its side-effects.

TODO: explain closures TODO: explain continuations TODO: explain monads (explicit about side-effects), what are effects TODO: explain the term ’pure’

In our research we selected Haskell as our functional programming language. [4]. The paper of [6] gives a comprehensive overview over the history of the language, how it developed and its features and is very interesting to read and get accustomed to the background of the language. A widely used introduction to programming in Haskell is [7]. The main points why we decided to go for Haskell are

- Pure, Lazy Evaluation, Higher-Order Functions and Static Typing - these are the most important points for the decision as they form the very foundation for composition, correctness, reasoning and verification.
- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications [5] [6] and is applicable to a number of real-world problems [8].

[3]Object-Oriented languages like Java let you to partially work around this limitation but are still far from *pure* functional programming.

[4]Although we did a bit of research using Scala (a mixed paradigm functional language) in ABS (see Appendix **??**), we deliberately ignored other functional languages as it is completely out-of-scope of this thesis to do an in-depth comparison of functional languages for their suitability to implement ABS.

[5]https://wiki.haskell.org/Applications_and_libraries

- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science e.g. parallelism & concurrency.
- In-house knowledge - the School of Computer Science of the University of Nottingham has a large amount of in-house knowledge in Haskell which can be put to use and leveraged in my thesis.

The main conclusion of the classical paper [9] is that *modularity* is the key to successful programming and can be achieved best using higher-order functions and lazy evaluation provided in functional languages like Haskell. The author argues that the ability to divide problems into sub-problems depends on the ability to glue the sub-problems together which depends strongly on the programming-language. He shows that laziness and higher-order functions are in combination a highly powerful glue and identifies this as the reason why functional languages are superior to structure programming. Another property of lazy evaluation is that it allows to describe infinite data-structures, which are computed as currently needed. This makes functions possible which produce an infinite stream which is consumed by another function - the decision of *how many* is decoupled from *how to*.

In the paper [10] Wadler describes Monads as the essence of functional programming (in Haskell). Originally inspired by monads from category-theory (see below) through the paper of Moggi [11], Wadler realized that monads can be used to structure functional programs [12]. A pure functional language like Haskell needs some way to perform impure (side-effects) computations otherwise it has no relevance for solving real-world problems like GUI-programming, graphics, concurrency,... . This is where monads come in, because ultimately they can be seen as a way to make effectful computations explicit [6]. In [10] Wadler shows how to factor out the error handling in a parser into monads which prevents code to be cluttered by cross-cutting concerns not relevant to the original problem. Other examples Wadler gives are the propagating of mutable state, (debugging) text-output during execution, non-deterministic choice. Further applications of monads are given in [10], [13], [14] where they are used for array updating, interpreting of a language formed by expressions in algebraic data-types, filters, parsers, exceptions, IO, emulating an imperative-style of programming. This seems to be exactly the way to go, tackling the problems mentioned in the introduction: making data-flow explicit, allowing to factor out cross-cutting concerns and encapsulate side-effects in types thus making them explicit. It may seem that one runs into efficiency-problems in a pure functional programming language when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of [15] showed that when approaching this problem from a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

The concept of monads was further generalized by Hughes in the concept of arrows [16]. The main difference between Monads and Arrows are that where monadic computations are parameterized only over their output-type, Arrows computations are parametrised both over their input- and output-type thus making Arrows more general. In [17] Hughes gives an example for the usage for Arrows in the field of circuit simulation. Streams are used to advance the simulation in discrete steps to calculate values of circuits thus the implementation is a form of *discrete event simulation* - which is in the direction we are heading already with ABS. As will be shown below, the concept of arrows is essential for Functional Reactive Programming a potential way to do ABS in pure functional programming.

B. Functional Reactive Programming

TODO: explain streams TODO: Yampa, BearRiver, Dunai

Functional Reactive Programming (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 frameworks which each has its own pro and contra. all started with fran, a domain specific language for graphics and animation and at yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa which is the reason why it was chosen as the FRP framework. Also, compared to other frameworks it does not distinguish between discrete and synchronous time but leaves that to the user of the framework how the time flow should be sampled (e.g. if the sampling is discrete or continuous - of course sampling always happens at discrete times but when we speak about discrete sampling we mean that time advances in natural numbers: 1,2,3,4,... and when speaking of continuous sampling then time advances in fractions of the natural numbers where the difference between each step is a real number in the range of [0..1])

[**?**] give a good overview of Yampa and FRP. Quote: "The essential abstraction that our system captures is time flow". Two *semantic* domains for progress of time: continuous and discrete.

[6]This is seen as one of the main impacts of Haskell had on the mainstream programming [6]

The first implementations of FRP (Fran) implemented FRP with synchronized stream processors which was also followed by [**?**]. Yampa is but using continuations inspired by Fudgets. In the stream processors approach "signals are represented as time-stamped streams, and signal functions are just functions from streams to streams", where "the Stream type can be implemented directly as (lazy) list in Haskell...":

"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" [**?**]. This quotation describes exactly one of the strengths using FRP in ACE

C. *Related Research*

III. 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 [18] 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. Still and because of its focus on technical details we can draw inspiration from MAS, how they define the concept of agents.

1) Agents: In MAS [19], [20] agents are 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 [19], [20] one distinguishes between different types of environments (based on [21]):

- 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 [19] 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. TODO: still look into how their definition and try to get inspiration for the ABS approach. Also give a short explanation. 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

IV. A FUNCTIONAL APPROACH

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

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, StateT in agents monad stack - pro-active read/write (Sugarscape): environment as agent, StateT in agents monad stack

care must be taken in case of agent-transactions: when aborting/refusing all changes to the environment must be rolled back =¿ instead of StateT use a transactional monad which allows us to revert changes to a save point at the start of the TX. if we drag the environment through all agents then we could easily revert changes but that then requires to hard-code the environment concept deep into the simulation scheduling/stepping which brings lots of inconveniences, also it would need us to fold the resulting multiple environments back into a single. If we had an environment-centric view then probably this is what we want but in ABS the focus is on the agents

question is if the TX sf runs in the same monad aw the agent or not. i opt for identity monad which prevents modification of the Environment in a transaction

also need to motivate the dt=0 in all TX processing: conceptually it all happens instantaneously (although arbitration is sequential) but agents must act time-sensitive

for environment we need transactional and shared state behaviour where we can have mutual exclusive access to shared data but also roll back changes we made. it should run deterministic when running agents not truly parallel. solution: run environment in a transactional state monad (TX monad). although the agents are executed in parallel in the end it (map) runs sequentially. this passes a mutable state through all agents which can act on it an roll back actions e.g. in case of a failed agent TX. if we dont need transactional behaviour then just use StateT monad. this ensures determinism. pro active environment is also easily possible by writing to the state. this approach behaves like sequential transactional although the agents run in parallel but how is this possible when using mapMSF ?

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

V. 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

In the course of the research for this paper we implemented the library *Chimera* [7] 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 C for a list of all the examples and the concepts used.

A. Example I: System Dynamics SIR Model

B. Example II: Agent-Based SIR Model

C. Example III: Discrete Event Simulation

generally a DES system is fixed in advance: the various DES objects are connected and communicate with each other through their ports. can we do this in our approach as well?

VI. 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. Hybrid ABS / Multi-method Approach

TODO: if we can do DES then its clearly a mult-method approach TODO: it is already a hybrid ABS as we can directly express SD as well

B. 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.

C. 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.

[7]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.

D. *Transactional Behaviour*

Imagine two agents A and B want to engage in a bartering process where agent A, is the seller who wants to sell an asset to agent B who is the buyer. Agent A sends Agent B a sell offer depending on how much agent A values this asset. Agent B receives this sell offer, checks if the price satisfies its utility, if it has enough wealth to buy the asset and replies with either a refusal or its own price offer. Agent A then considers agent Bs offer and if it is happy it replies to agent B with an acceptance of the offer, removes the asset from its inventory and increases its wealth. Agent B receives this acceptance offer, puts the asset in its inventory and decreases its wealth (note that this process could involve a potentially arbitrary number of steps without loss of generality). We can see this behaviour as a kind of multi-step transactional behaviour because agents have to respect their budget constraints which means that they cannot spend more wealth or assets than they have. This implies that they have to 'lock' the asset and the amount of cash they are bartering about during the bartering process. If both come to an agreement they will swap the asset and the cash and if they refuse their offers they have to 'unlock' them. In classic OO implementations it is quite easy to implement this as normally only one agent is active at a time due to sequential (discrete event scheduling approach) scheduling of the simulation. This allows then agent A which is active, to directly interact with agent B through method calls. The sequential updating ensures that no other agent will touch the asset or cash and the direct method calls ensure a synchronous updating of the mutable state of both objects with no time passing between these updates.

Unfortunately this is not directly possible using our approach. The reasons for this are the following: - an agent cannot access another agents' state or invoke its signal-function directly - due to the parallel scheduling all agents act virtually at the same time - each agent-agent interaction takes time =¿ this leads to the problem that an agent could get engaged with many sell offers at the same time which it could all individually satisfy but not all of them together. Worse: each interaction requires time which could lead to changed wealth and then turning down of offers. In the end it boils down to the problem of losing the transactional behaviour as it is possible in an OO approach.

A potential solution could be 1. agents get the possibility to freeze time which means that the SFs are still evaluated as before but with a time-delta of 0 2. when an agent wants to start a transactional behaviour it freezes time and initiates a data-flow to the receiving agent and switches into a waiting-behaviour. 3. if the model is implemented in a way that a receiving agent could receive an arbitrary number of such data-flows it can only process them one-by-one which means it must store them in a list and transact them one-by-one. 4. finally this requires all agents need to distinguish between transactional data-flows and time-dependent ones. This means that an agent should check for Sell Offers *in every step, independent of the time-delta* but when they want to make moves they must execute their action only every 1.0 time-units. When freezing time this will ensure that they won't act.

VII. 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

- 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. Backus, "Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359579

[2] A. Church, *The Calculi of Lambda-conversion*. Princeton University Press, 1941, google-Books-ID: KCOuGztKVgcC.

[3] G. Michaelson, *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation, 2011, google-Books-ID: gKvwPtvsSjsC.

[4] B. J. MacLennan, *Functional Programming: Practice and Theory*. Addison-Wesley, Jan. 1990, google-Books-ID: JqhQAAAAMAAJ.

[5] C. Allen and J. Moronuki, *Haskell Programming from First Principles*. Allen and Moronuki Publishing, Jul. 2016, google-Books-ID: 5FaXDAEACAAJ.

[6] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A History of Haskell: Being Lazy with Class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–1–12–55. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238856

[7] G. Hutton, *Programming in Haskell*. Cambridge University Press, Aug. 2016, google-Books-ID: 1xHPDAAAQBAJ.

[8] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[9] J. Hughes, "Why Functional Programming Matters," *Comput. J.*, vol. 32, no. 2, pp. 98–107, Apr. 1989. [Online]. Available: http://dx.doi.org/10.1093/comjnl/32.2.98

[10] P. Wadler, "The Essence of Functional Programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '92. New York, NY, USA: ACM, 1992, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/143165.143169

[11] E. Moggi, "Computational Lambda-calculus and Monads," in *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Piscataway, NJ, USA: IEEE Press, 1989, pp. 14–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=77350.77353

[12] P. Wadler, "Comprehending Monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90. New York, NY, USA: ACM, 1990, pp. 61–78. [Online]. Available: http://doi.acm.org/10.1145/91556.91592

[13] ——, "Monads for Functional Programming," in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995, pp. 24–52. [Online]. Available: http://dl.acm.org/citation.cfm?id=647698.734146

[14] ——, "How to Declare an Imperative," *ACM Comput. Surv.*, vol. 29, no. 3, pp. 240–263, Sep. 1997. [Online]. Available: http://doi.acm.org/10.1145/262009.262011

[15] C. Okasaki, *Purely Functional Data Structures*. New York, NY, USA: Cambridge University Press, 1999.

[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] ——, "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

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

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

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

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

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

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

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

[25] 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

[26] 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

[27] 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

[28] 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
Full code of the System Dynamics SIR implementation

```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
```

Appendix B
Full code of the agent-based SIR implementation

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

type SIRAgentState = SIRState

type SIREnvironment = [AgentId]

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

type SIREventSource = EventSource SIRAgentState SIRMsg SIREnvironment

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

contactRate :: Double
contactRate = 5

illnessDuration :: Double
illnessDuration = 15

contactSS :: Int
contactSS = 20

illnessTimeoutSS :: Int
illnessTimeoutSS = 2

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

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

    return (adefsSusceptible ++ adefsInfected, agentIds)

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

    return adef

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

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

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

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

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

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

sirAgentInfectedBehaviour :: RandomGen g => g -> SIRAgentBehaviourIgnoreEnv
sirAgentInfectedBehaviour g = proc ain -> do
    ao' <- doOnce (setAgentState Infected) -< agentOutFromIn ain
    returnA -< respondToContactWith Infected ain ao'

-- RECOVERED
sirAgentRecovered :: SIRAgentBehaviour
sirAgentRecovered = doOnceR $ setAgentStateR Recovered

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

--------------------------------------------------------------------------------
runSIR :: IO ()
runSIR = do
    -- parallel strategy, no updating/folding of environment, no shuffling, rng-seed of 42
    params <- initSimulation Parallel Nothing Nothing False (Just 42)
    (initAdefs, initEnv) <- createSIRNumInfected agentCount numInfected
    let dynamics = simulateAggregateTime initAdefs initEnv params dt t aggregate
    print dynamics

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

APPENDIX C
EXAMPLES

In this appendix we give a list of all the examples we have implemented and discuss implementation details relevant [8]. 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 [22]. 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 [9].

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 [23].

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 [24] 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 [10].

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

[9]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.

[10]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 [25], [26] 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 [27] 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 [28]. 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 [11]: "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

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