Towards pure functional Agent-Based Simulation in Haskell

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

TODO: use SocialForce model as application, its an excellent use-case. Also we have working implementations in AnyLogic and RePast already which allows for awesome comparability!

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

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

In this paper we challenge these suggestions and ask how agent-based simulation can be implemented in the pure functional programming language Haskell.

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

In [5] Axelrod reports the vulnerability of ABS to misunderstanding. Due to informal specifications of models and change-requests among members of a research-team bugs are very likely to be introduced. He also reported how difficult it was to reproduce the work of [6] which took the team four months which was due to inconsistencies between the original code and the published paper. The consequence is that counter-intuitive simulation results can lead to weeks of checking whether the code matches the model and is bug-free as reported in [3]. As ABS is almost always used for scientific research, producing often break-through scientific results as pointed out in [5] and used for policy making, these ABS need to be *free of bugs*, *verified against their specification*, *validated against hypotheses* and ultimately be *reproducible*.

Pure functional programming in Haskell claims [7], [8] to overcome these problems or at least allows to tackle them more effectively due to its declarative nature, free of side-effects, strong static type system.

TODO: aim TODO: objective TODO: outcome TODO: contribution

A. Costy bugs due to language features

[ ] knight capital glitch [ ] mars lander [ ] moon landing [ ] ? [ ] ethereum & blockchain technology

- The 3 major benefits of the approach I claim 1. code == spec 2. can rule out serious class of bugs 3. we can perform reasoning about the simulation in code need to be metricated: e.g. this is really only possible in Haskell and not in Java. This needs thorough thinking about which metrics are used, how they can be aquired, how they can be compared,...

- I NEED TO SHOW HOW I CAN MAKE HASKELL RELEVANT IN THE FIELD OF ABS -¿ as far as I know so far no reasoning has been done in the way I intend to do it in the field of ABS. My hypothesis is that it is really only possible in Haskell due to its explicit side-effects, type-system, declarative style,... -¿ TODO: need to check if this is really unique to haskell -¿ the functional-reactive approach seems to bring a new view to ABS with an embedded language for explicit time-semantics. Together with parallel/sequential updating this allows implementing System-Dynamics and agents which rely on continuous time-semantics e.g. SIR-Agents. Maybe i invented a hybrid between SD and ABS? Also what about time-traveling? The problem is that this is not really clear as i hypothesize that is completely novel approach to ABS - again I need to check this! -¿ TODO: is this really unique to functional reactive? E.g. what about Repast, NetLogo, AnyLogic, other Java-Frameworks? -¿ maybe i have to admit that its not as unique as thought

In General i need to show that - Haskells general benefits & drawbacks over other Languages in the Field of ABS (e.g. Java, NetLogo, Repast) e.g. declarative style, reasoning, explicit about side-effects, performance, difficult to reason about performance, space-leaks difficult. So this focuses on the general comparison between the established technologies of ABS and Haskell but not yet on Haskells suitability in comparison to these other technologies. Here we talk about reasoning, side-effects, performance IN GENERAL TERMS, NOT SPECIFIC TO ABS. We need to distinguish between -¿ general technicalities e.g. lambda-calculus (denotational formalism) or turing-machine (operational formalism) foundations, declarative style, lazy-evaluation allows to split the producer from the consumer, explicit about side-effects, not possible for in-order updates,... -¿ and in what they result e.g. fewer lines of code, ruling out of bugs, reasoning, lower performance, difficult to reason about space-time

- Haskells suitability to implement ABS in comparison to other languages and technologies in the Field. Here the focus is on general problems in ABS and how they can and are solved using Haskell e.g. send message, changing environment, handling of time, replications, parallelism/concurrency,...

- Why using Haskell in ABS - do the general benefits / drawbacks apply equally well? Are there unique advantages? Can we do things in Haskell which are not possible in other technologies or just very hard? E.g. the hybrid-approach I created with FRP: how unique is it e.g. can other technologies easily implement it as well? Other potential advantages: recursive simulation. Here we DO NOT concentrate on general technicalities but see how they apply when using it for ABS and if they create a unique benefit for Haskell in ABS.

i need to show that different programming languages and paradigms have different power and are differently well suited to specific problems: the ultimate claim i need to show is that haskell is more powerful than java or C++ - the question is if this also makes it superior in applying it to problems: being more powerful, can all problems of java be solved better in haskell as well? this is i believe not the case e.g. gui- or game- programming. the question then is: what is the power of a programming language? can we measure it?

so what i need to show is how well haskell and its power are suited for implementing ABS. does the fact that haskell is much more powerful than existing technologies in ABS lead to the point that it is better suited for ABS? in fact it is power vs. better suited

B. The power of a language

[ ] more expressive: we can express complex problems more directly and with less overhead. note that this is domain-specifix: the mechanisms of a language allow to create abstractions which solve the domain-specific problem. the better these mechanisms support one in this task, the more powerful the language is in the given domain. now we end up by defining what ”better” support means [ ] one could in pronciple do system programming in haskell by provoding bindings to code written in c and / or assembly but when the program is dominated by calls to these bindings then one could as well work directly in these lower languages and saves one from the overhead of the bindings [ ] but very often a domain consists of multiple subdomains. [ ] my hypothesis is that haskell is not well suited for domains which are dominated by managing and manipulating a global mutable state through side-effects / effectful computations. examples are gui-programming and computer games (state spread accross GPU and cpu, user input,...). this does not mean that it is not possible to implememt these things in haskell (it has been done with some sucess) but that the solution becomes too complex at some point. [ ] conciesness [ ] low ceremony [ ] susceptibility to bugs [ ] verbosity [ ] reasoning about performance [ ] reasoning about space requirements

C. Measuring a language

Define scientific measures: e.g. Lines Of Code (show relation to Bugs & Defects, which is an objective measure: http://www.stevemcconnel
https://softwareengineering.stackexchange.com/questions/185660/is-the-average-number-of-bugs-per-loc-the-same-for-different-programmi
Book: Code Complete, https://www.mayerdan.com/ruby/2012/11/11/bugs-per-line-of-code-ratio), also experience reports by

companies which show that Haskell has huge benefits when applied to the same domain of a previous implementation of a different language, post on stack overflow / research gate / reddit, read experience reports from http://cufp.org/2015/ Also need to show the problem of operational reasoning as opposed to denotational reasoning

D. The Abstraction Hierarchy

1st: Functional vs. Object Oriented 2nd: Haskell vs. Java 3rd: FrABS vs. Repast

II. Background

A. Agent-Based Simulation

We understand ABS as a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is of knowledge. Those parts, called agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. So the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in a network of neighbours by exchange of messages [9]. It is important to note that we focus our understanding of ABS on a very specific kind of agents where the focus is on communicating entities with individual, localized behaviour from out of which the global behaviour of the system emerges. We informally assume the following about our agents:

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

B. Implementation

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

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

C. SIR Model

- do not introduce SIR model in that length, also don't discuss SD and ABS, only minimal definition of what we understand as ABS, ignore definition of SD completely - good introduction to pure functional programming in Haskell: this is VERY difficult as it is a VAST topic where one can get lost quickly. focus on the central concepts: no assignment, recursion, pattern matching, static type-system with higher-kinded polymorphism - focus on the benefits of the pure functional approach -¿ program looks very much like a specification -¿ can rule out bugs at compile time -¿ can guarantee reproducibility at compile time -¿ 2 update-strategies without the need of different -¿ testing using quickcheck, testing = writing program spec -¿ reasoning: TODO

D. 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 libraries which each has its benefits and deficits. The very first functional reactive language was Fran, a domain specific language for graphics and animation. At Yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa, the most recent FRP library [10]. The essence of FRP with Yampa is that one describes the system in terms of signal functions in a declarative manner using the EDSL of Yampa. During execution the top level signal functions will then be evaluated and return new signal functions which act as continuations. A major design goal for FRP is to free the programmer

from 'presentation' details by providing the ability to think in terms of 'modeling'. It is common that an FRP program is concise enough to also serve as a specification for the problem it solves [11].

Yampa has been used in multiple agent-based applications: [12] uses Yampa for implementing a robot-simulation, [13] implement the classical Space Invaders game using Yampa, [14] implements a Pong-clone, the thesis of [15] shows how Yampa can be used for implementing a Game-Engine, [16] implemented a 3D first-person shooter game with the style of Quake 3 in Yampa. Note that although all these applications don't focus explicitly on agents all of them inherently deal with kinds of agents which share properties of classical agents: game-entities, robots,... Other fields in which Yampa was successfully used were programming of synthesizers, network routers, computer music development and has been successfully combined with monads [17].

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

E. NetLogo

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

III. Reasoning

[ ] reasoning: sd is always reproducible because all runs outside the io moand. this means that potential RNG seeds are always the same and do not depend on outside states e.g. time [ ] for the agents this means that repeated runs with the same seed are guaranteed to result in the same dynamics as there are again no possibilities to introduce e.g. random seeds or values from the outside which may change between runs like time, files,...

-¿ spatial and network behaviour is EXACTLY the same except selection of neighbours =¿ what can we reason about it regarding the dynamics?

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

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

i need to get a deep understanding in writing correct code and reasoning about correctness in Haskell - look into papers: https://wiki.haskell.org/Research_papers/Testing_and_correctness https://www.reddit.com/r/haskell/comments/4tagq3/examples_of_realworld_haskell_usage_where/ https://stackoverflow.com/questions/4077970/can-haskell-functions-be-proved-model-checked-verified-with-correctne

reasoning about equivalence between SD and ABS implementation in the same framework

IV. Testing

TODO: property-based and unit testing of a model

TODO: modular testing of agents

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

[18]

-¿ testing replies: an infected agent always replies with contact infected, a recovered never replies, a susceptible never replies -¿ testing contacts: a susceptible contacts occaaionally, an infected contacts occasionally, a recovered never contacts. -¿ testing: an infected agent recovering after (same as falling ball?) -¿ testing: a susceptible agent gets infected after infect contact -¿ testing: can we measure the occasional distribution to verify?

testing a functional reactive agent-based simulation using quickcheck and unit-tests

V. Time-Traveling

[19]

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

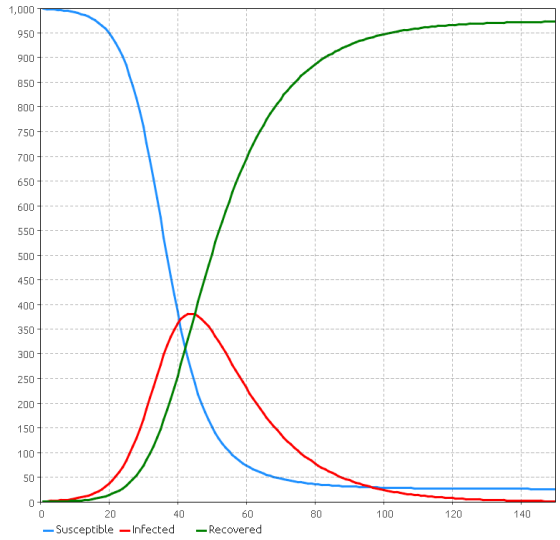


Fig. 1: Dynamics of the SIR model using the SD approach generated with AnyLogic Personal Learning Edition 8.1.0. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150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

VI. Equivalence of SD and ABS implementation

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

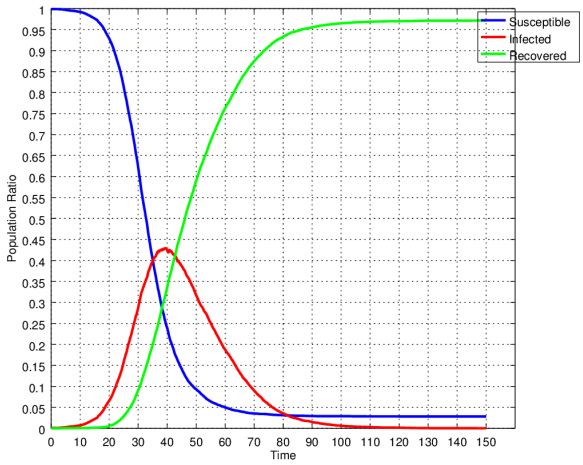
A. Correctness of SD implementation

The mathematical formulas for susceptible, infected an recovered stocks with the infection- and recover-rates are directly translated into the SD implementation as seen in Appendix B. We can thus assume that the translation from the mathematical definition to Haskell code is correct as it *is* exactly the same. This leaves us with the question how the values of the stocks and flows are distributed between each other. When solving the mathematical equations numerically, one is dividing time into infinitely small intervals and calculates the new value of each formula at each time-interval at the same time: the values update all at the same time. The same needs to happen in our SD implementation which is indeed the case: *runSD* as seen in line 127 uses *simulateTime* behind the scenes with the *parallel* update-strategy which uses a *map* to iterate over all agents. This implies by definition of map that all stocks and flows, which are in fact agents, update virtually at the same time. Due to pureness of runSD we can rule out side-effects which could only occur when running in the IO Monad or using unsafePerformIO [1]. It is clear now that the values update at the same time and calculate the correct values as defined in the mathematical formulas. The distribution of the values happens by using absolute stock- and flow-ids and it is easy to check that the ids used in *flowInFrom/flowOutTo* and *stockInFrom/stockOutTo* build the correct connections as seen in the Stock-And-Flow diagram in Figure **??**. This leaves us with the implementation of *integral*. Theoretically we arrive at the correct mathematical solution if we use infinitely small Δt but this is of course impossible using a computer. When looking at the implementation of *integral* it becomes apparent that it uses the rectangle-rule. TODO: need more details and look a bit into theory of numerically solving integral (e.g. classic newton, or runge-kutta). The rectangle rule needs small Δt for sufficient accuracy as we have shown in Figure 3. For our reference dynamics in Figure **??** we used $\Delta t = 0.001$, which computes 150,000 steps. When comparing it with dynamics as seen in Figure 1 generated by the professional software package AnyLogic Personal Learning Edition 8.1.0 we can safely say that a $\Delta t = 0.001$ is sufficiently small to generate matching accuracy.

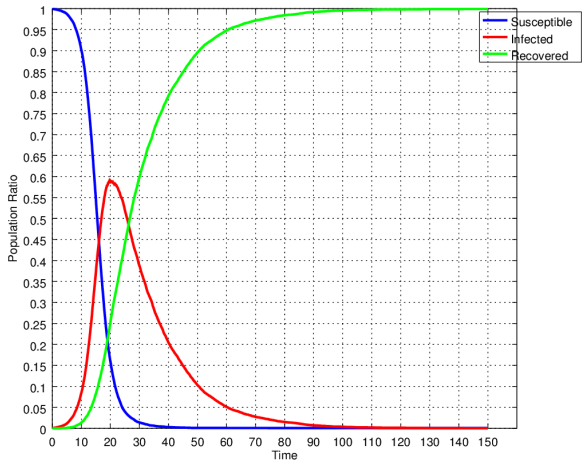
B. Correctness of ABS implementation

The question is now what *correctness* of an ABS implementation means. This is indeed a very non-trivial problem and is highly dependent on the model. In our case of the agent-based SIR implementation we define correctness to mean *qualitatively approximating the SD dynamics*. This means that to show that our ABS implementation is correct, we need to show that it is

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



(a) Infected agents make contact.



(b) Susceptible and Infected agents make contact.

Fig. 2: Dynamics of different contact policies. Using same model parameters as in Figure **??**.

equivalent to the SD implementation. TODO: this is a bit short, maybe we can say more about this. need to read a few papers and get better understanding on this topic

Also we ask the question: why is it that the susceptible agents must make the contact and the infected only replying? can't just the infected agents make contact thus saving a round-trip? also why are not both making contact? When comparing the two contact-policies as seem in Figure 2, it is clear that both don't match neither the SD dynamics of Figure **??** and ABM dynamics of Figure **??**.

C. Equivalence

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

VII. Results

VIII. Discussion

A. Emulating System Dynamics

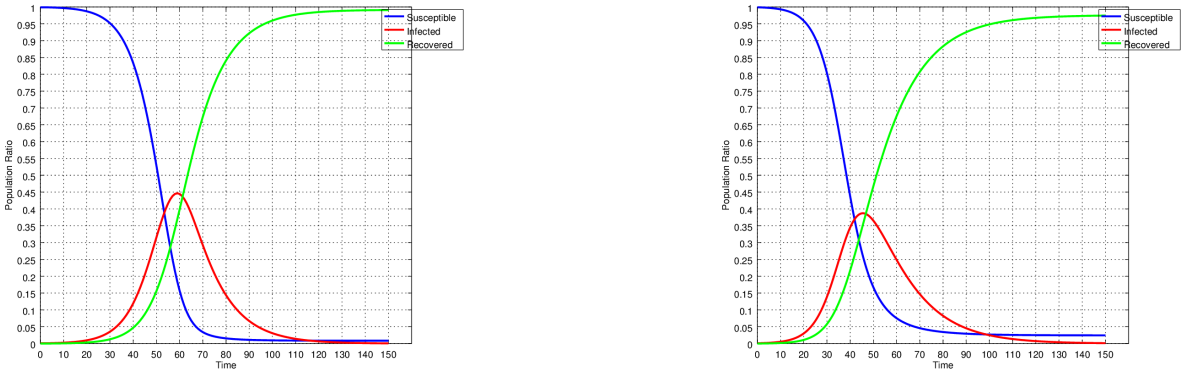
Due to the continuous time-semantics which can be expressed in our agent-based approach we can also emulate SD. Every stock and flow is then just an agent which exchange messages where the simulation is stepped using the *parallel* update-strategy. We add wrappers and type-definitions for convenience which increases the expressivity of the code, resembling system dynamics specifications. As a proof-of-concept we emulated the system dynamics of the SIR model, the code can be seen in Appendix B. Note that the code really looks like a SD specification with the integrals of the mathematical specification directly showing up in the code - the implementation is correct per definition. Due to the internal implementation of the *integral* function of Yampa which uses the rectangle-rule to integrate, one must ensure to sample the system dynamics with small enough Δt as becomes apparent in Figure 3.

B. Recursive ABS

Due to the inherent recursive nature of functional programming we came up with the idea of *recursive* ABS in which agents can recursively run the simulation within the simulation which would allow them to project their own actions into the future. So far it only exists as a proof-of-concept and we are currently only aware of a single model [20] in the field of ABS which does recursive simulation. The implementation of recursive ABS is very natural due to the explicit data-flow and lack of side-effects which eases the task very much. Unfortunately we cannot go into detail of our approach as it is beyond the scope of the paper.

C. Different Agent-Types

[ ] can we implement different types of agents interacting with each other in the same simulation ? with different behaviour funcs, digferent state? yes, also not possible in NetLogo to my knowledge. but they must have the same messages, emvironment



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

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

D. Advantages

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

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

E. Disadvantages

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

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

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

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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

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

[6] ——, "The Convergence and Stability of Cultures: Local Convergence and Global Polarization," Santa Fe Institute, Working Paper, Mar. 1995. [Online]. Available: http://econpapers.repec.org/paper/wopsafiwp/95-03-028.htm

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

[8] P. Hudak and M. Jones, "Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity," Department of Computer Science, Yale University, New Haven, CT, Research Report YALEU/DCS/RR-1049, Oct. 1994.

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

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

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

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

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

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

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

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

[17] I. Perez, M. Brenz, and H. Nilsson, "Functional Reactive Programming, Refactored," in *Proceedings of the 9th International Symposium on Haskell*, ser. Haskell 2016. New York, NY, USA: ACM, 2016, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2976002.2976010

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

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

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

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

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

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

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

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

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

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 [1]. The model description itself has no real time-semantics, the agents act in every time-step. Only the environment may change its behaviour after a given number of steps but this is easily expressed without time-semantics as described in the model by Epstein & Axtell [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 [21].

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

C. Schelling Segregation

This is an implementation of [22] with extended agent-behaviour which allows to study dynamics of different optimization behaviour: local or global, nearest/random, increasing/binary/future. This is also the only 'real' model in which the recursive features were applied [4].

[2]The examples are freely available under https://github.com/thalerjonathan/phd/tree/master/coding/libraries/frABS/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 [23], [24] and TODO: cite my own paper on update-strategies.

TODO: implement

E. Heroes & Cowards

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

TODO: implement

F. SIRS

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

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

G. Reactive SIRS

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

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

H. System Dynamics SIR

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

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

I. WildFire

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

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

J. Double Auction

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

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

K. Policy Effects

This is an implementation of a model inspired by Uri Wilensky [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

```
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
```