Implementing System Dynamics

A pure functional correct-by-construction approach in Haskell

JONATHAN THALER, University of Nottingham, United Kingdom

TODO: select journals - ACM Transactions on Modeling and Computer Simulation (TOMACS): https://tomacs.acm.org/ - ?

In this short paper we investigate how to implement System Dynamics in the functional programming language Haskell. We use the concept of Functional Reactive Programming which allows to express continuous- and discrete-time systems in a functional way. We show that System Dynamics map very natural to Functional Reactive Programming. Together with Haskell's strong static type system, we arrive at a correct-by-construction implementation which deterministic reproducibility we can guarantee at compile-time.

Additional Key Words and Phrases: System Dynamics, Functional Reactive Programming, Haskell

ACM Reference Format:
Jonathan Thaler. 2019. Implementing System Dynamics: A pure functional correct-by-construction approach in Haskell. 1, 1 (July 2019), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

There exists a large number of simulation packages which allow the convenient creation of System Dynamics simulations by straight-forward visual diagram creation. One simply creates stocks and flows, connects them, specifies the flow-rates and initial parameters and then runs them. An example for such a visual diagram creation in the simulation package AnyLogic can be seen in Figure 1.

The aim of this paper is to look into how System Dynamics can be implemented in raw code without the use of a simulation package. Our language of choice is Haskell because TODO.

We use the well known SIR model [2] from epidemiology to demonstrate our approach.

The contribution of the paper is the demonstration of how a correct-by-construction System Dynamics simulation can be implemented using Haskell.

2 RELATED WORK

TODO: is there some?

3 SIR MODEL

We introduce the SIR model as a motivating example and use-case for our implementation. It is a very well studied and understood compartment model from epidemiology [2] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles [1] spreading through a population. In this model, people in a population of size N can be in either one of three states *Susceptible*, *Infected* or *Recovered* at a particular time, where it is assumed that initially there is at least one infected person in the population. People interact *on average* with a given rate of β other people per time-unit and become infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time-units and is then immune to further infections. An interaction between infected persons does not lead to

Author's address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk, University of Nottingham, 7301 Wollaton Rd, Nottingham, NG8 1BB, United Kingdom.

2019. XXXX-XXXX/2019/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

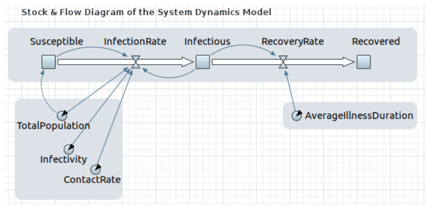

Fig. 1. Visual System Dynamics Diagram in AnyLogic Personal Learning Edition 8.3.1.

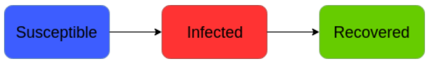

Fig. 2. States and transitions in the SIR compartment model.

re-infection, thus these interactions are ignored in this model. This definition gives rise to three compartments with the transitions as seen in Figure 2.

Before looking into how one can simulate this model in an agent-based approach we first explain how to formalize it using System Dynamics (SD) [3]. In SD one models a system through differential equations, allowing to conveniently express continuous systems which change over time. The advantage of a SD solution is that one has an analytically tractable solution against which e.g. agent-based solutions can be validated. The problem is that the more complex a system, the more difficult it is to derive differential equations describing the global system, to a point where it simply becomes impossible. This is the strength of an agent-based approach over SD, which allows to model a system when only the constituting parts and their interactions are known but not the macro behaviour of the whole system. As will be shown later, the agent-based approach exhibits further benefits over SD.

The dynamics of the SIR model can be formalized in SD with the following equations:

TODO: there seems to be an unnerving space after the f letters, can we get rid of them?

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -infectionRate \tag{1}$$

$$\frac{\mathrm{d}I}{\mathrm{d}t} = infectionRate - recoveryRate \tag{2}$$

$$\frac{\mathrm{d}R}{\mathrm{d}t} = recoveryRate \tag{3}$$

$$infectionRate = \frac{I \beta S \gamma}{N} \tag{4}$$

$$recoveryRate = \frac{I}{\delta} \tag{5}$$

Solving these equations is done by integrating over time. In the SD terminology, the integrals are called *Stocks* and the values over which is integrated over time are called *Flows*. At $t = 0$ a single agent is infected because if there wouldn't be any infected agents, the system would immediately reach equilibrium - this is also the formal definition of the steady state of the system: as soon as $I(t) = 0$ the system won't change any more.

TODO: there seems to be an unnerving space after the f letters, can we get rid of them?

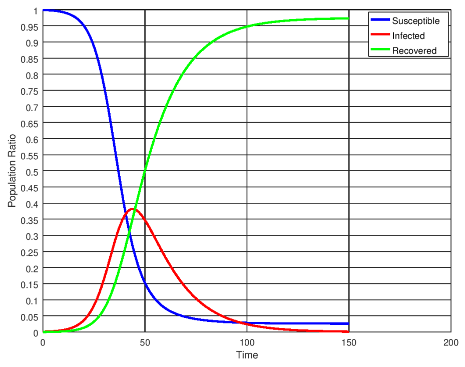

Fig. 3. Dynamics of the SIR compartment model. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

$$S(t) = N - I(0) + \int_0^t -infectionRate \, \mathrm{d}t \tag{6}$$

$$I(0) = 1 \tag{7}$$

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

$$R(t) = \int_0^t recoveryRate \, \mathrm{d}t \tag{9}$$

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

4 A CORRECT-BY-CONSTRUCTION IMPLEMENTATION

In this section we develop a correct-by-construction implementation step-by-step. The complete code is attached in Appendix A. Note that the constant parameters *populationSize, infectedCount, contactRate, infectivity, illnessDuration* are defined globally and omitted for clarity.

Computing the dynamics of a SD model happens by integrating the time over the equations. So conceptually we treat our SD model as an continuous function which is defined over time = 0 -> infinity and at each point in time outputs the values of each stock. In the case of the SIR model we have 3 stocks: Susceptible, Infected and Recovered. Thus we start our implementation by defining the output of our SD function: for each time-step we have the values of the 3 stocks:

```haskell
type SIRStep = (Time, Double, Double, Double)
```

Next we define our continuous SD function which we obviously make a signal function. It has no input, because a SD system is only defined in its own terms and parameters without external input and has as output the *SIRStep*. Thus we define the following function type:

```haskell
sir :: SF () SIRStep
```

An SD model is fundamentally built on feedback: the values at time t depend on the history. Thus we introduce feedback in which we feed the last step into the next step. Yampa provides the *loopPre :: c → SF (a, c) (b, c) → SF a b* function for that. It takes an initial value and a feedback signal function which receives the input *a* and the previous (or initial) value of the feedback and has to return the output *b* and the new feedback value *c*. *loopPre* then returns simply a signal function from *a* to *b* with the feedback happening transparent in the feedback signal function. Our initial feedback value is the initial state of the SD model at $t = 0$. Further we define the type of the feedback signal function:

```
sir = loopPre (0, initSus, initInf, initRec) sirFeedback
  where
    initSus = populationSize - infectedCount
    initInf = infectedCount
    initRec = 0

    sirFeedback :: SF ((), SIRStep) (SIRStep, SIRStep)
```

The next step is to implement the feedback signal function. As input we get *(a, c)* where *a* is the empty tuple () because a SD simulation has no input, and *c* is the fed back *SIRStep* from the previous (initial) step. With this we have all relevant data so we can implement the feedback function. We first match on the tuple inputs and construct a signal function using *proc*:

```
    sirFeedback = proc (_, (_, s, i, _)) -> do
```

Now we define our flows which are *infection rate* and *recovery rate*. The formulas for both of them can be seen in equations TODO (refer to the differential equations). This directly translates into Haskell code:

```
      let infectionRate = (i * contactRate * s * infectivity) / populationSize
          recoveryRate  = i / illnessDuration
```

Next we need to compute the values of the three stocks, following the formulas of TODO (refer to the Integral formulas). For this we need the *integral* function of Yampa which integrates over a numerical input using the rectangle rule. Adding initial values can be achieved with the (^<<) operator of arrowized programming. This directly translates into Haskell code:

```
      s' <- (initSus+) ^<< integral -< (-infectionRate)
      i' <- (initInf+) ^<< integral -< (infectionRate - recoveryRate)
      r' <- (initRec+) ^<< integral -< recoveryRate
```

We also need the current time of the simulation. For this we use Yampas *time* function:

```
      t <- time -< ()
```

Now we only need to return the output and the feedback value. Both types are the same thus we simply duplicate the tuple:

```
      returnA -< dupe (t, s', i', r')

    dupe :: a -> (a, a)
    dupe a = (a, a)
```

We want to run the SD model for a given time with a given Δt by running the *sir* signal function. To *purely* run a signal function Yampa provides the function *embed :: SF a b -> (a, [(DTime, Maybe a)]) -> [b]* which allows to run an SF for a given number of steps where in each step one provides the Δt and an input *a*. The function then returns the output of the signal function for each step. Note that the input is optional, indicated by *Maybe*. In the first step at $t = 0$, the initial *a* is applied and whenever the input is *Nothing* in subsequent steps, the last *a* which was not *Nothing* is re-used.

```
runSD :: Time -> DTime -> [SIRStep]
runSD t dt = embed sir ((), steps)
  where
    steps = replicate (floor (t / dt)) (dt, Nothing)
```

5 DISCUSSION

TODO: argue why it is correct-by-construction, why reproducible guaranteed at compile-time,... support our initial hypothesis and claims from introduction

TODO: integral is the fundamental function - need to show that it indeed implements the rectangle rule. - show that with too large dt we arrive at slightly different results after same time-steps - implement a better integral function using better behaved numerical integration

6 CONCLUSION

TODO: wow its so super

ACKNOWLEDGMENTS

The authors would like to thank

REFERENCES

[1] Richard H. Enns. 2010. *It's a Nonlinear World* (1st ed.). Springer Publishing Company, Incorporated.

[2] W. O. Kermack and A. G. McKendrick. 1927. A Contribution to the Mathematical Theory of Epidemics. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 115, 772 (Aug. 1927), 700–721. https://doi.org/10.1098/rspa.1927.0118

[3] Donald E. Porter. 1962. Industrial Dynamics. Jay Forrester. M.I.T. Press, Cambridge, Mass.; Wiley, New York, 1961. xv + 464 pp. Illus. $18. *Science* 135, 3502 (Feb. 1962), 426–427. https://doi.org/10.1126/science.135.3502.426-a

A COMPLETE CODE

This appendix presents the complete code [1] which was introduced step-by-step in Section 4.

```haskell
populationSize :: Double
populationSize = 1000

infectedCount :: Double
infectedCount = 1

contactRate :: Double
contactRate = 5

infectivity :: Double
infectivity = 0.05

illnessDuration :: Double
illnessDuration = 15

type SIRStep = (Time, Double, Double, Double)

sir :: SF () SIRStep
sir = loopPre (0, initSus, initInf, initRec) sirFeedback
  where
    initSus = populationSize - infectedCount
    initInf = infectedCount
    initRec = 0

    sirFeedback :: SF ((), SIRStep) (SIRStep, SIRStep)
    sirFeedback = proc (_, (_, s, i, _)) -> do
      let infectionRate = (i * contactRate * s * infectivity) / populationSize
          recoveryRate  = i / illnessDuration

      t <- time -< ()

      s' <- (initSus+) ^<< integral -< (-infectionRate)
      i' <- (initInf+) ^<< integral -< (infectionRate - recoveryRate)
      r' <- (initRec+) ^<< integral -< recoveryRate

      returnA -< dupe (t, s', i', r')

    dupe :: a -> (a, a)
    dupe a = (a, a)

runSD :: Time -> DTime -> [SIRStep]
runSD t dt = embed sir ((), steps)
  where
    steps = replicate (floor (t / dt)) (dt, Nothing)
```

Received May 2018

[1]The complete project, including visualisation and exporter to Matlab, is freely available on the Git Repository https://github.com/thalerjonathan/phd/tree/master/public/sdhaskell/SIR