Implementing Correct-By-Construction System Dynamics

A pure functional approach

JONATHAN THALER, University of Nottingham, United Kingdom

TODO: write related work

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-time systems in a functional way. We show that System Dynamics map naturally to the abstractions provided by 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 Correct-By-Construction System Dynamics: A pure functional approach. 1, 1 (July 2019), 9 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 the model. An example for such a visual diagram creation in the simulation package AnyLogic can be seen in Figure 1.

Still, implementing System Dynamics directly in code is not as straight forward and involves numerical integration which can be quite tricky to get right. Thus, the aim of this paper is to look into how System Dynamics models can be implemented in code correctly without the use of a simulation package. We use the well known SIR model [7] from epidemiology to demonstrate our approach.

Our language of choice is Haskell because it emphasises a declarative programming style in which one describes *what* instead of *how* to compute. Further it allows to rule out interference with non-deterministic influences or side-effects already at compile-time. This is of fundamental importance for System Dynamics because it behaves completely deterministic and involves no stochastics or non-determinism whatsoever. Also, we make use of Functional Reactive Programming which allows to express continuous-time systems in a functional way.

We show that by this approach we can arrive at correct-by-construction implementations of System Dynamic models. This means that the correctness of the code is obvious because we have closed the gap between the model specification and its implementation. Thus, the contribution of the paper is the demonstration of how to implement correct-by-construction System Dynamics simulations using Haskell and Functional Reactive Programming.

2 RELATED WORK

TODO: is there some?

TODO: simulink & matlab

TODO: TODO:

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

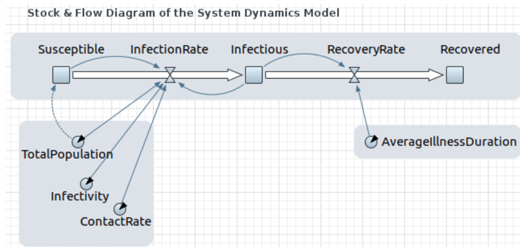

Fig. 1. Visual System Dynamics Diagram of the SIR model in AnyLogic Personal Learning Edition 8.3.1.

3 BACKGROUND

In this section we give a short overview of the concepts used throughout the paper.

3.1 Haskell

We are using the functional programming language Haskell. The paper of [4] 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. The main points why we decided to go for Haskell are:

- Rich Feature-Set - it has all fundamental concepts of the pure functional programming paradigm of which we explain the most important below.
- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications [4], is applicable to a number of real-world problems [9] and has a large number of libraries available [1].
- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science. Further, the community is the main source of high-quality libraries.

3.2 Functional Reactive Programming

Functional Reactive Programming is a way to implement systems with continuous and discrete time-semantics in pure functional languages. There are many different approaches and implementations but in our approach we use *Arrowized* FRP [5, 6] as implemented in the library Yampa [1, 3, 8].

The central concept in Arrowized FRP is the Signal Function (SF) which can be understood as a *process over time* which maps an input- to an output-signal. A signal can be understood as a value which varies over time. Thus, signal functions have an awareness of the passing of time by having access to Δt which are positive time-steps with which the system is sampled.

$$Signal\ \alpha \approx Time \rightarrow \alpha$$
$$SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$$

Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow to change system behaviour in case of events, run signal functions and generate stochastic events and random-number streams. We shortly discuss the relevant combinators and concepts we use throughout the paper. For a more in-depth discussion we refer to [1, 3, 8].

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

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

3.3 Arrowized programming

Yampa's signal functions are arrows, requiring us to program with arrows. Arrows are a generalisation of monads which, in addition to the already familiar parameterisation over the output type, allow parameterisation over their input type as well [5, 6].

In general, arrows can be understood to be computations that represent processes, which have an input of a specific type, process it and output a new type. This is the reason why Yampa is using arrows to represent their signal functions: the concept of processes, which signal functions are, maps naturally to arrows.

There exists a number of arrow combinators which allow arrowized programing in a point-free style but due to lack of space we will not discuss them here. Instead we make use of Paterson's do-notation for arrows [10] which makes code more readable as it allows us to program with points.

To show how arrowized programming works, we implement a simple signal function, which calculates the acceleration of a falling mass on its vertical axis as an example [11].

```
fallingMass :: Double -> Double -> SF () Double
fallingMass p0 v0 = proc _ -> do
  v <- arr (+v0) <<< integral -< (-9.8)
  p <- arr (+p0) <<< integral -< v
  returnA -< p
```

To create an arrow, the *proc* keyword is used, which binds a variable after which the *do* of Patersons do-notation [10] follows. Using the signal function *integral :: SF v v* of Yampa which integrates the input value over time using the rectangle rule, we calculate the current velocity and the position based on the initial position *p0* and velocity *v0*. The <<< is one of the arrow combinators which composes two arrow computations and *arr* simply lifts a pure function into an arrow. To pass an input to an arrow, -< is used and <- to bind the result of an arrow computation to a variable. Finally to return a value from an arrow, *returnA* is used.

To hide the Δt from the type-signature of a signal function Yampa uses a clever trick. All signal functions SF are defined for t = 0 and then change their signature into SF' which actually gets passed Δt as its first parameter. This happens transparently to the user of the library and is supported by not publicly exporting SF'.

4 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 [7] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles [2] 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 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.

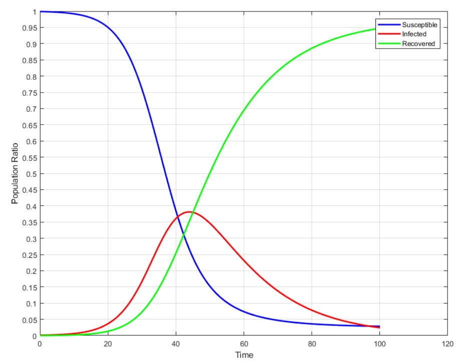

Fig. 3. Dynamics of the SIR compartment model. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run until $t = 150$. Plot generated from data by this Haskell implementation using Octave.

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

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -infectionRate \tag{1}$$
$$\frac{\mathrm{d}I}{\mathrm{d}t} = infectionRate - recoveryRate \tag{2}$$
$$\frac{\mathrm{d}R}{\mathrm{d}t} = recoveryRate \tag{3}$$

$$infectionRate = \frac{I \beta S \gamma}{N} \tag{4}$$
$$recoveryRate = \frac{I}{\delta} \tag{5}$$

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.

$$S(t) = N - I(0) + \int_0^t -infectionRate \, \mathrm{d}t \tag{6}$$
$$I(0) = 1 \tag{7}$$
$$I(t) = \int_0^t infectionRate - recoveryRate \, \mathrm{d}t \tag{8}$$
$$R(t) = \int_0^t recoveryRate \, \mathrm{d}t \tag{9}$$

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

5 A CORRECT-BY-CONSTRUCTION IMPLEMENTATION

In this section we step-by-step develop a correct-by-construction implementation. 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 a 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:

```
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:

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

An SD model is fundamentally built on feedback: the values at time t depend on the previous step. 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 -< ()
```

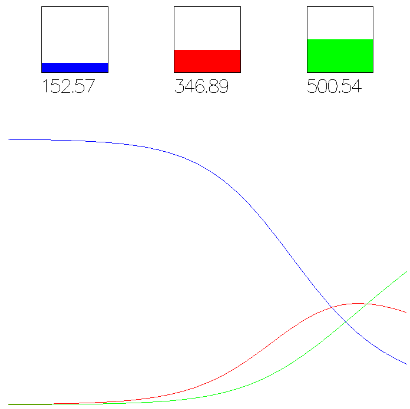

Fig. 4. Snapshot of a real-time visualisation of a SIR compartment model simulating using Haskell at $t = 50$. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent.

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)
```

6 DISCUSSION

We claim that our implementation is correct-by-construction because the code *is* the model specification - we have closed the gap between the specification and its implementation. Also we can guarantee that no non-deterministic influences can happening, neither in our nor Yampas library code due to the strong static type system of Haskell. This guarantees that repeated runs of the simulation will always result in the exact same dynamics given the same initial parameters, something of fundamental importance in System Dynamics. See Figure 4 for a real-time visualisation of the SIR simulation of our Haskell implementation.

Although we have translated our model specifications directly into code we still we need to validate the dynamics and test the system for its numerical behaviour under varying Δt. This is necessary because numerical integration, which happens in the *integral* function, can be susceptible to instability and errors. Yampa implements the simple rectangle-rule of numerical integration which requires very small Δt to keep the errors minimal and arrive at sufficiently good results.

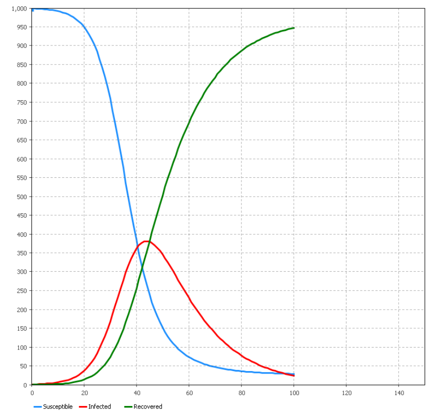

Fig. 5. System Dynamics simulation of SIR compartment model in AnyLogic Personal Learning Edition 8.3.1. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run until $t = 100$.

We have run the simulation with varying Δt to show what difference varying Δt can have on the simulation dynamics. We ran the simulations until $t = 100$ with a population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ and initially 1 infected agent. For comparison we looked at the final values at $t = 100$ of the susceptible, infected and recovered stocks. Also we compare the time and the value when the infected stock reaches its maximum. The values are reported in the Table 6.

Δt	Susceptibles	Infected	Recovered	Max Infected
1.0	17.52	26.87	955.61	419.07 @ t = 51
0.5	23.24	25.63	951.12	399.53 @ t = 47.5
0.1	27.56	24.27	948.17	384.71 @ t = 44.7
$1e-2$	28.52	24.11	947.36	381.48 @ t = 43.97
$1e-3$	28.62	24.08	947.30	381.16 @ t = 43.9
AnyLogic	28.625	24.081	947.294	381.132 @ t = 44

As additional validation we added the results of a System Dynamics simulation in AnyLogic Personal Learning Edition 8.3.1, which is reported in the last row in the Table 6. Also we provided a visualisation of the AnyLogics simulation dynamics in Figure 5. By comparing the results in Table 6 and the dynamics in Figure 5 to 3 we can conclude that we arrive at the same dynamics, validating the correctness of our simulation also against an existing, well-known and established System Dynamics software package.

7 CONCLUSION

In this paper we have shown how to implement System Dynamics in a way that the resulting implementation is correct-by-construction, where the gap between the formal model specifications and the actual implementation in code is closed. We used the pure functional programming language Haskell for it and built on the Functional Reactive Programming concept to express our continuous-time simulation. The provided abstractions of Haskell and Functional Reactive Programming allowed to close the gap between the specification and implementation and further guarantee the absence of non-deterministic influences already at compile-time, making our correct-by-construction claims even stronger.

Further we showed the influence of different Δt and validated our implementation against the industry-strength System Dynamics simulation package AnyLogic Personal Learning Edition 8.3.1 where we could match our results with the one of AnyLogic, proving the correctness of our system also on the dynamics level.

Obviously the numerical well behaviour depends on the integral function which uses the rectangle rule. We showed that for the SIR model and small enough Δt, the rectangle rule works well enough. Still it might be of benefit if we provide more sophisticated numerical integration like Runge-Kutta methods. We leave this for further research.

The key strength of System Dynamic simulation packages is their visual representation which allows non-programmers to express System Dynamics models and simulate them. We believe that one can auto-generate Haskell code using our approach to implement System Dynamics from such diagrams but leave this for further research.

Also we are very well aware that due to the vast amount of visual simulation packages available for System Dynamics, there is no big need for implementing such simulations directly in code. Still we hope that our pure functional approach with Functional Reactive Programming might spark an interest in approaching the implementation of System Dynamics from a new perspective, which might lead to pure functional back-ends of visual simulation packages, giving them more confidence in their correctness.

ACKNOWLEDGMENTS

The authors would like to thank J. Hey for constructive feedback, comments and valuable discussions.

REFERENCES

[1] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/871895.871897

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

[3] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, Johan Jeuring and Simon L. Peyton Jones (Eds.). Number 2638 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[4] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[5] John Hughes. 2000. Generalising Monads to Arrows. *Sci. Comput. Program.* 37, 1-3 (May 2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[6] John Hughes. 2005. Programming with Arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming (AFP'04)*. Springer-Verlag, Berlin, Heidelberg, 73–129. https://doi.org/10.1007/11546382_2

[7] 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

[8] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

[9] Bryan O'Sullivan, John Goerzen, and Don Stewart. 2008. *Real World Haskell* (1st ed.). O'Reilly Media, Inc.

[10] Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/507635.507664

[11] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 2:1–2:27. https://doi.org/10.1145/3110246

A COMPLETE CODE

This appendix presents the complete code [2] which was introduced step-by-step in Section 5.

```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

[2] 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