Verification & correctness of Agent-Based Simulation

JONATHAN THALER, THORSTEN ALTENKIRCH, and PEER-OLAF SIEBERS, University of Nottingham, United Kingdom

Additional Key Words and Phrases: Verification, Dependent Types, Agent-Based Simulation, Discrete Event Simulation

ACM Reference Format:
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Verification & correctness of Agent-Based Simulation. *Proc. ACM Program. Lang.* 9, 4, Article 39 (September 2018), 8 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Previous research has shown that the pure functional programming paradigm as in Haskell is very suitable to implement agent-based simulations. Building on FRP and MSFs the work developed an elegant implementation of an agent-based SIR model which was pure. By statically removing all external influences of randomness already at compile time through types, this guarantees that repeated simulation runs with the same starting conditions will always result in the same dynamics - guaranteed at compile time. This previous research focused only on establishing the basic concepts of ABS in functional programming but it did not explore the inherent strength of functional programming for verification and correctness any further than guaranteeing the reproducibility of the simulation at compile time.

This paper picks up where the previous research has left and wants to investigate the usefulness of pure and dependently typed functional programming for verification and correctness of agent-based simulation. We are especially interested if requirements of an ABS can be guaranteed on a stronger level by those paradigms, if a larger class of bugs can be excluded already at compile time and whether we can express model properties and invariants already at compile time on a type level. Further we are interested in how far we can reason about an agent-based model in a dependently typed implementation. Also we investigate the use of QuickCheck for code- and model-testing.

the key points to investigate in this report are: - testing of abs: unit- & property testing - using types for invariants / bug free - reasoning about correctness in code - reasoning about dynamics in code

Validation & Verification in ABS http://www2.econ.iastate.edu/tesfatsi/VVAccreditationSimModels.OBalci1998.pdf : verification = are we building the model right? validation = are we building the right model?

good paper http://www2.econ.iastate.edu/tesfatsi/VVAccreditationSimModels.OBalci1998.pdf : very nice 15 guidelines and life cycles, VERY valuable for background and introduction

Authors' address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk; Thorsten Altenkirch, thorsten.altenkirch@nottingham.ac.uk; Peer-Olaf Siebers, University of Nottingham, 7301 Wollaton Rd, Nottingham, NG8 1BB, United Kingdom, peer-olaf.siebers@nottingham.ac.uk.

2475-1421/2018/9-ART39 $15.00
https://doi.org/0000001.0000001

http://www2.econ.iastate.edu/tesfatsi/VVSimulationModels.JKleijnen1995.pdf : suggests good programming practice which is extremely important for high quality code and reduces bugs but real world practice and experience shows that this alone is not enough, even the best programmers make mistakes which often can be prevented through a strong static or a dependent type system already at compile time. What we can guarantee already at compile time, doesn't need to be checked at run-time which saves substantial amount of time as at run-time there may be a huge number of execution paths through the simulation which is almost always simply not feasible to check (note that we also need to check all combinations). This paper also cites modularity as very important for verification: divide and conquer and test all modules separately. this is especially easy in functional programming as composability is much better than with traditional oop due to the lack of interdependence between data and code as in objects and the lack of global mutable state (e.g. class variables or global variables) - this makes code extremely convenient to test. The paper also discusses statistical tests (the t test) to check if the outcome of a simulation is sufficiently close to real-world dynamics. Also the paper suggests using animations to visualise the processes within the simulation for verification purposes (of course they note that animation may be misleading when one focuses on too short simulation runs).

good paper:https://link.springer.com/chapter/10.1007/978-3-642-01109-2_10 -> verification. "This is essentially the question: does the model do what we think it is supposed to do? Whenever a model has an analytical solution, a condition which embraces almost all conventional economic theory, verification is a matter of checking the mathematics." -> validation: "In an important sense, the current process of building ABMs is a discovery process, of discovering the types of behavioural rules for agents which appear to be consistent with phenomena we observe." => can we encode phenomena we observe in the types? can we use types for the discovery process as well? can dependent types guide our exploratory approach to ABS? -> "Because such models are based on simulation, the lack of an analytical solution (in general) means that verification is harder, since there is no single result the model must match. Moreover, testing the range of model outcomes provides a test only in respect to a prior judgment on the plausibility of the potential range of outcomes. In this sense, verification blends into validation."

either one has an analytical model as the basis of an agent-based model (ABM) or one does not. In the former case, e.g. the SIR model, one can very easily validate the dynamcis generated by the ABM to the one generated by the analytical solution (e.g. through System Dynamics). Of course the dynamics wont be exactly the same as ABS discretisizes the approach and introduces stochastics which means, one must validate averaged dynamics. In the latter case one has basically no idea or description of the emergent behaviour of the system prior to its execution. It is important to have some hypothesis about the emergent property / dynamics. The question is how verification / validation works in this setting as there is no formal description of the expected behaviour: we don't have a ground-truth against which we can compare our simulation dynamics. (eventuell hilft hier hans vollbrecht weiter: Simulation hat hier den Sinn, die Controller anhand der Roboteraufgabe zu validieren, Bei solchen Simulationen ist man interessiert an allen mÃűglichen Sequenzen, und da das meist zu viele sind, an einer mÃűglichst gut verteilten Stichprobenmenge. Hier geht es weniger um richtige Zeitmodellierung, sondern um den Test aller mÃűglichen Ereignissequenzen.)

look into DEVS

TODO: the implementation phase is just one stage in a longer process http://jasss.soc.surrey.ac.uk/12/1/1.html

WE FOCUS ON VERIFICATION important: we are not concerned here with validating a model with the real world system it simulates. this is an entirely different problem and focuses on the questions if we have built the right model. we are interested here in extremely strong verification: have we built the model right? we are especially interested in to which extend purely and dependently-typed functional programming can support us in this task.

http://jasss.soc.surrey.ac.uk/8/1/5.html: "For some time now, Agent Based Modelling has been used to simulate and explore complex systems, which have proved intractable to other modelling approaches such as mathematical modelling. More generally, computer modelling offers a greater flexibility and scope to represent phenomena that do not naturally translate into an analytical framework. Agent Based Models however, by their very nature, require more rigorous programming standards than other computer simulations. This is because researchers are cued to expect the unexpected in the output of their simulations: they are looking for the 'surprise' that shows an interesting emergent effect in the complex system. It is important, then, to be absolutely clear that the model running in the computer is behaving exactly as specified in the design. It is very easy, in the several thousand lines of code that are involved in programming an Agent Based Model, for bugs to creep in. Unlike mathematical models, where the derivations are open to scrutiny in the publication of the work, the code used for an Agent Based Model is not checked as part of the peer-review process, and there may even be Intellectual Property Rights issues with providing the source code in an accompanying web page."

http://jasss.soc.surrey.ac.uk/12/1/1.html: "a prerequisite to understanding a simulation is to make sure that there is no significant disparity between what we think the computer code is doing and what is actually doing. One could be tempted to think that, given that the code has been programmed by someone, surely there is always at least one person - the programmer - who knows precisely what the code does. Unfortunately, the truth tends to be quite different, as the leading figures in the field report, including the following: You should assume that, no matter how carefully you have designed and built your simulation, it will contain bugs (code that does something different to what you wanted and expected), "Achieving internal validity is harder than it might seem. The problem is knowing whether an unexpected result is a reflection of a mistake in the programming, or a surprising consequence of the model itself. [âĂę] As is often the case, confirming that the model was correctly programmed was substantially more work than programming the model in the first place. This problem is particularly acute in the case of agent-based simulation. The complex and exploratory nature of most agent-based models implies that, before running a model, there is some uncertainty about what the model will produce. Not knowing a priori what to expect makes it difficult to discern whether an unexpected outcome has been generated as a legitimate result of the assumptions embedded in the model or, on the contrary, it is due to an error or an artefact created in the model design, its implementation, or its execution."

general requirements to ABS - modelling progress of time (steward robinson simulation book, chapter 2) - modelling variability (steward robinson simulation book, chapter 2) - fixing random number streams to allow simulations to be repeated under same conditions (steward robinson simulation book, chapter 1.3.2 and chapter 2) - only rely on past -> solved with Arrowized FRP - bugs due to implicitly mutable state -> can be ensured by pure functional programming - ruling out external sources of non-determinism / randomness -> can be ensured by pure functional programming - correct interaction protocols -> can be ensured by dependent state machines - deterministic time-delta -> TODO: can we ensure it through dependent-types at type-level? - repeated runs lead to same dynamics -> can be ensured by pure functional programming

steward robinson simulation book bulletpoints - chapter 8.2: speed of coding, transparency, flexibility, run-speed - chapter 8.3: three activities - 1 coding, 2 testing verification and white-box validating, 3 documenting - chapter 9.7: nature of simulation: terminating vs. non-terminating - chapter 9.7: nature of simulation output: transient or steady-state (steady-state cycle, shifting steady-state) steward robinson simulation book on implementation - meaning of implementation -> 1 implementing the findings: conduct a study which defines and gathers all findings about the model and document them -> 2 implementing the model -> 3 implementing the learning

steward robinson simulation book on verification, validation and confidence - Verification is the process of ensuring that the model design has been transformed into a computer model with sufficient accuracy (Davis 1992) - Validation is the process of ensuring that the model is sufficiently accurate for the purpose at hand (Carson 1986). - Verification has a narrow definition and can be seen as a subset of the wider issue of validation - In Verification and validation the aim is to ensure, that the model is sufficiently accurate, which always implies its purpose. - => the purpose / objectives mus be known BEFORE it is validated - white-box validation: detailed, micro check if each part of the model represent the real world with sufficient accuracy -> intrinsic to model coding - black-box validation: overall, macro check whether the model provides a sufficiently accurate representation of the real world system -> can only be performed once model code is complete - other definition of verification: it is a test of the fidelity with which the conceptual model is converted into the computer model - verification (and validation) is a continuous process => if it is already there in the programming language / supported by it e.g. through types,... then this is much easier to do - difficulties of verification and validation -> there is no such thing as general validity: a model should be built for one purpose as simple as possible and not be too general, otherwise it becomes too bloated and too difficult / impossible to analyse -> there may be no real world to compare against: simulations are developed for proposed systems, new production / facilities which dont exist yet. -> which real world?: the real world can be interpreted in different ways => a model valid to one person may not be valid to another -> often the real world data are inaccurate -> there is not enough time to verify and validate everything -> confidence, not validity: it is not possible to prove that a model is valid, instead one should think of confidence in its validity. => verification and validation is thus not the proof that a model is correct but trying to prove that the model is incorrect, the more tests/checks one carries out which show that it is NOT incorrect, the more confidence we can place on the models validity - methods of verification and validation -> conceptual model validation: judment based on the documentation -> data validation: analysing data for inconsistencies -> verification and white-box validation -> both conceptually different but often treated together because both occur continuously through model coding -> what should be checked: timings (cycle times, arrival times,...), control of elements (breakdown frequency, shift patterns), control flows (e.g. routing), control logic (e.g. scheduling, stock replenishment), distribution sampling (samples obtained from an empirial distribution) -> verification and whilte-box validation methods -> checking code: reading through code and ensure right data and logic is there. explain to others/discuss together/others should look at your code. -> Visual checks -> inspecting output reports

-> black-box testing: consider overall behaviour of the model without looking into its parts, basically two ways -> comparison with the real system: statistical tests -> comparison with another model (e.g. mathematical equations): could compare exactly or also through statistical tests ->

2 TESTING / VERIFICATION

TODO: explore ABS testing in pure functional Haskell - we need to distinguish between two types of testing/verification -> 1. testing/verification of models for which we have real-world data or an analytical solution which can act as a ground-truth. examples for such models are the SIR model, stock-market simulations, social simulations of all kind -> 2. testing/verification of models which are just exploratory and which are only be inspired by real-world phenomena. examples for such models are Epsteins Sugarscape and Agent_Zero

2.1 Black Box Verification

Defined as treating the functionality to test as a black box with inputs and outputs and comparing controlled inputs to expected outputs.

In Black Box Verification one generally feeds input and compares it to expected output. In the case of ABS we have two things to black-box test:

(1) Agent Behaviour - test isolated agent behaviour under given inputs using unit- and property-based testing
(2) Simulation Dynamics - compare emergent dynamics of the ABS as a whole under given inputs to an analytical solution / real-world dynamics in case there exists some using statistical tests
(3) Hypotheses- test whether hypotheses are valid / invalid using unit- and property-based testing. TODO: how can we formulate hypotheses in unit- and/or property-based tests?

- testing of the final dynamics: how close do they match the analytical solution - can we express model properties in tests e.g. quickcheck? - property-testing shines here - isolated tests: how easy can we test parts of an agent / simulation?

2.1.1 Comparison of dynamics against existing data. - utilise a statistical test with H0 "ABS and comparison is not the same" and H1 "ABS and comparison is the same" - how many replications and how do we average? - which statistical test do we implement? (steward robinson simulation book, chapter 12.4.4) -> Normalizsed Mean Squared Error (NMSE) -> TODO: implement confidence interval -> TODO: what about chi-squared? -> TODO: what about paired-t confidence interval

IMPORTANT: this is not what we are after here in this paper, statistical tests are a science on their own and there actually exists quite a large amount of literature for conducting statistical tests on ABS dynamics: Robinson Book (TODO: find additional literature)

2.2 White Box Verification

Defined as directly looking at the code and reasoning that the code does really what it has to implement. - coverage testing

3 VERIFICATION OF SIR

In this section we verify our agent-based implementation of the SIR model. Verification of our implementation should be fairly straight-forward and easy as the model is given in differential equations which gives us a formal specification of the model which we can use directly in our verification process. We will also conduct white-box verification and for one property it will be the only way of ensuring that our model is correct as we cannot guarantee it through black-box verification.

3.1 Black Box Verification

3.1.1 Agent Behaviour. When conducting black-box testing for the SIR model, we test if the *isolated* behaviour of an agent in all three states Susceptible, Infected and Recovered, corresponds to model specifications. The crucial thing though is that we are dealing with a stochastic system where the agents act *on averages*, which means we need to average our tests as well.

The interface of the agent behaviours are defined below. When running the SF with a given Δt one has to feed in the state of all the other agents as input and the agent outputs its state it is after this Δt.

```
data SIRState
  = Susceptible
  | Infected
  | Recovered

type SIRAgent = SF [SIRState] SIRState

susceptibleAgent :: RandomGen g => g -> SIRAgent
infectedAgent :: RandomGen g => g -> SIRAgent
recoveredAgent :: SIRAgent
```

Susceptible Behaviour. A susceptible agent *may* become infected, depending on the number of infected agents in relation to non-infected the susceptible agent has contact to. To make this property testable we run a susceptible agent for 1.0 time-unit (note that we are sampling the system with small Δt e.g. 0.1) and then check if it is infected - that is it returns infected as its current state. Obviously we need to pay attention to the fact that we are dealing with a stochastic system thus we can only talk about averages and thus it does not suffice to only run a single agent but we are repeating this for e.g. $N = 10.000$ agents (all with different RNGs). We then need a formula for the required fraction of the N agents which should have become infected on average. Per 1.0 time-unit, a susceptible agent makes *on average* contact with β other agents where in the case of a contact with an infected agent the susceptible agent becomes infected with a given probability γ. In this description there is another probability hidden, which is the probability of making contact with an infected agent which is simply the ratio of number of infected agents to number not infected agents. The formula for the target fraction of agents which become infected is then: $\beta * \gamma * \frac{numberofinfected}{numberofnon-infected}$. To check whether this test has passed we compare the required amount of agents which on average should become infected to the one from our tests (simply count the agents which got infected and divide by N) and if the value lies within some small ϵ then we accept the test as passed.

Obviously the input to the susceptible agents which we can vary is the set of agents with which the susceptible agents make contact with. To save us from constructing all possible edge-cases and combinations and testing them with unit-tests we use QuickCheck which creates them randomly for us and reduces them also to all relevant edge-cases. This is an example for how to use property-based testing in ABS where QuickCheck can be of immense help generating random test-data to cover all cases.

TODO: derive the target-fraction formula from the differential equations TODO: can we encode this somehow on a type level using dependent types? then we don't need to test this property any more

Infected Behaviour. An infected agent *will always* recover after a finite time, which is *on average* after δ time-units. Note that this property involves stochastics too, so to test this property we run a large number of infected agents e.g. $N = 10.000$ (all with different RNGs) until they recover, record the time of each agents recovery and then average over all recovery times. To check whether this test has passed we compare the average recovery times to δ and if they lie within some small ϵ then we accept the test as passed. We use QuickCheck in this case as well to generate the set of other agents as input for the infected agents. Strictly speaking this would not be necessary as an infected agent never makes contact with other agents and simply ignores them - we could as well just feed in an empty list. We opted for using QuickCheck for the following reasons:

- We wanted to stick to the interface specification of the agent-implementation as close as possible which asks to pass the states of all agents as input.

- We shouldn't make any assumptions about the actual implementation and if it REALLY ignores the other agents, so we strictly stick to the interface which requires us to input the states of all the other agents.
- The set of other agents is ignored when determining whether the test has failed or not which indicates by construction that the behaviour of an infected agent does not depend on other agents.
- We are not just running a single replication over 10.000 agents but 100 of them which should give black-box verification more strength.

TODO: derive the average formula from the differential equations TODO: can we encode this somehow on a type level using dependent types? then we don't need to test this property any more

Recovered Behaviour. A recovered agent will stay in the recovered state *forever*. Obviously we cannot write a black-box test that truly verifies that because it had to run in fact forever. In this case we need to resort to White Box Verification (see below).

Because we use multiple replications in combination with QuickCheck obviously results in longer test-runs (about 5 minutes on my machine) In our implementation we utilized the FRP paradigm. It seems that functional programming and FRP allow extremely easy testing of individual agent behaviour because FP and FRP compose extremely well which in turn means that there are no global dependencies as e.g. in OOP where we have to be very careful to clean up the system after each test - this is not an issue at all in our *pure* approach to ABS.

3.1.2 Simulation Dynamics. We won't go into the details of comparing the dynamics of an ABS to an analytical solution, that has been done already by [**?**]. What is important is to note that population-size matters: different population-size results in slightly different dynamics in SD => need same population size in ABS (probably...?). Note that it is utterly difficult to compare the dynamics of an ABS to the one of a SD approach as ABS dynamics are stochastic which explore a much wider spectrum of dynamics e.g. it could be the case, that the infected agent recovers without having infected any other agent, which would lead to an extreme mismatch to the SD approach but is absolutely a valid dynamic in the case of an ABS. The question is then rather if and how far those two are *really* comparable as it seems that the ABS is a more powerful system which presents many more paths through the dynamics. TODO: i really want to solve this for the SIR approach -> confidence intervals? -> NMSE? -> does it even make sense?

3.2 White Box Verification

White-Box verification is necessary when we need to reason about properties like *forever*, *never*, which cannot be guaranteed from black-box tests. In the case of the SIR model we have the following invariants:

- A susceptible agent will *never* make the transition to recovered.
- An infected agent will *never* make the transition to susceptible.
- A recovered agent will *forever* stay recovered.

All these invariants can be guaranteed when reasoning about the code. An additional help will be then coverage testing with which we can show that an infected agent never returns susceptible, and a susceptible agent never returned infected given all of their functionality was covered which has to imply that it can never occur!

Lets start with looking at the recovered behaviour as it is the simplest one. We then continue with the infected behaviour and end with the susceptible behaviour as it is the most complex one.

Recovered Behaviour. The implementation of the recovered behaviour is as follows:

```
recoveredAgent :: SIRAgent
recoveredAgent = arr (const Recovered)
```

Just by looking at the type we can guarantee the following:

- it is pure, no side-effects of any kind can occur
- no stochasticity possible because no RNG is fed in / we don't run in the random monad

The implementation is as concise as it can be and we can reason that it is indeed a correct implementation of the recovered specification: we lift the constant function which returns the Recovered state into an arrow. Per definition and by looking at the implementation, the constant function ignores its input and returns always the same value. This is exactly the behaviour which we need for the recovered agent. Thus we can reason that the recovered agent will return Recovered *forever* which means our implementation is indeed correct.

Infected Behaviour.

```
infectedAgent :: RandomGen g => g -> SIRAgent
infectedAgent g =
    switch
      infected
      (const recoveredAgent)
  where
    infected :: SF [SIRState] (SIRState, Event ())
    infected = proc _ -> do
      recEvt <- occasionally g illnessDuration () -< ()
      let a = event Infected (const Recovered) recEvt
      returnA -< (a, recEvt)
```

Susceptible Behaviour.

```
susceptibleAgent :: RandomGen g
                 => g
                 -> SIRAgent
susceptibleAgent g =
    switch
      (susceptible g)
      (const (infectedAgent g))
  where
    susceptible :: RandomGen g => g -> SF [SIRState] (SIRState, Event ())
    susceptible g = proc as -> do
      makeContact <- occasionally g (1 / contactRate) () -< ()

      -- NOTE: strangely if we are not splitting all if-then-else into
      -- separate but only a single one, then it seems not to work,
      -- dunno why
      if isEvent makeContact
        then (do
          a <- drawRandomElemSF g -< as
          case a of
            Just Infected -> do
              i <- randomBoolSF g infectivity -< ()
              if i
```

```
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _       -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

4 VERIFICATION OF SUGARSCAPE

Sugarscape is an exploratory model inspired by real-world phenomenon which means it has lots of hypotheses implicit in the model but there does not exist real-world data / dynamics against which one could validate the simulated dynamics. Still we can conduct black-box verification because we have an informal model specification but we cannot do any statistical testing of simulated dynamics as we don't have data acting as ground-truth. But what we can do and what we will explore extensively in this section is how we can encode hypotheses about the dynamics (prior to running the simulation) in unit- and property-based tests and check them. Obviously white-box verification applies as well because we can reason about the code whether it matches the informal model specification or not.

4.1 Black Box Verification

4.1.1 Agent Behaviour.

4.1.2 Hypotheses.

4.2 White Box Verification

5 TOWARDS DEPENDENTLY TYPED VERIFICATION

Independent of the programming paradigm, there exist fundamentally two approaches implementing agent-based simulation: time- and event-driven. In the time-driven approach, the simulation is stepped in fixed Δt and all agents are executed at each time-step - they act virtually in lock-step at the same time. The approach is inspired by the theory of continuous system dynamics (TODO: cite). In the event-driven approach, the system is advanced through events, generated by the agents, and the global system state changes by jumping from event to event, where the state is held constant in between. The approach is inspired by discrete event simulation (DES) (TODO: citation) which is formalized in the DEVS formalism [?].

In a preceding paper we investigated how to derive a time-driven pure functional ABS approach in Haskell (TODO: cite my paper). We came to quite satisfactory results and implemented also a number of agent-based models of various complexity (TODO: cite schelling, sugarscape, agent zero). Still we identified weaknesses due to the underlying functional reactive programming (FRP) approach. It is possible to define partial implementations which diverge during runtime, which may be difficult to determine for complex models for a programmer at compile time. Also sampling the system with fixed Δt can lead to severe performance problems when small Δt are required, as was shown in our paper. The later problem is well known in the simulation community and thus as a remedy an event-driven approach was suggested [?]. In this paper for the first time, we derive a pure functional event-driven agent-based simulation. Instead of using Haskell, which provides already libraries for DES [?], we focus on the dependently typed pure functional programming language Idris. In our previous paper we hypothesised that dependent types may offer interesting new insights and approaches to ABS but it was unclear how exactly we can make use of them, which was left for further research. In this paper we hypothesise that, as opposed to a time-driven approach, the even-driven approach is especially suited to make proper use of dependent types due to its different nature. Note that both a pure functional event-driven approach to ABS *and* the use of dependent types in ABS has so far never been investigated, which is the unique contribution of this paper. If we can construct a dependently typed program of the SIR ABM which is total, then we have a proof-by-construction that the SIR model reaches a steady-state after finite time

Dependent Types are the holy grail in functional programming as they allow to express even stronger guarantees about the correctness of programs and go as far where programs and types become constructive proofs [?] which must be total by definition [?], [?], [?], [?]. Thus the next obvious step is to apply them to our pure functional approach of agent-based simulation. So far no research in applying dependent types to agent-based simulation exists at all and it is not clear whether dependent types do make sense in this setting. We explore this for the first time and ask more specifically how we can add dependent types to our pure functional approach, which conceptual implications this has for ABS and what we gain from doing so. Note that we can only scratch the surface and lay down basic ideas and leave a proper in-depth treatment of this topic for further research. We use Idris [?], [?] as language of choice as it is very close to Haskell with focus on real-world application and running programs as opposed to other languages with dependent types e.g. Agda and Coq which serve primarily as proof assistants.

Dependent Types promise the following:

(1) Types as proofs - In dependently types languages, types can depend on any values and are first-class objects themselves. TODO: make more clear
(2) Totality and termination - Constructive proofs must terminate, this means a well-typed program (which is itself a proof) is always terminating which in turn means that it must consist out of total functions. A total function is defined by [?] as: it terminates with a well-typed result or produces a non-empty finite prefix of a well-typed infinite result in finite time. Idris is turing complete but is able to check the totality of a function under some circumstances but not in general as it would imply that it can solve the halting problem. Other dependently typed languages like Agda or Coq restrict recursion to ensure totality of all their functions - this makes them non turing complete.

dependent-types: -> encode model-invariants on a meta-level -> encode dynamics (what? feedbacks? positive/negative) on a meta-level -> totality equals steady-state of a simulation, can enforce totality if required through type-level programming -> probabilistic types can encode probability distributions in types already about which we can then reason -> can we encode objectives in types? -> agents as dependently typed continuations?: need a dependently typed concept of a process over time

5.1 Dependently Typed SIR

Intuitively, based upon our model and the equations we can argue that the SIR model enters a steady state as soon as there are no more infected agents. Thus we can informally argue that a SIR model must always terminate as:

(1) Only infected agents can infect susceptible agents.
(2) Eventually after a finite time every infected agent will recover.
(3) There is no way to move from the consuming *recovered* state back into the *infected* state [1].

Thus a SIR model must enter a steady state after finite steps / in finite time.

This result gives us the confidence, that the agent-based approach will terminate, given it is really a correct implementation of the SD model. Still this does not proof that the agent-based approach itself will terminate and so far no proof of the totality of it was given. Dependent Types and Idris ability for totality and termination checking should theoretically allow us to proof that an

[1]There exists an extended SIR model, called SIRS which adds a cycle to the state-machine by introducing a transition from recovered to susceptible but we don't consider that here.

agent-based SIR implementation terminates after finite time: if an implementation of the agent-based SIR model in Idris is total it is a proof by construction. Note that such an implementation should not run for a limited virtual time but run unrestricted of the time and the simulation should terminate as soon as there are no more infected agents. We hypothesize that it should be possible due to the nature of the state transitions where there are no cycles and that all infected agents will eventually reach the recovered state. Abandoning the FRP approach and starting fresh, the question is how we implement a *total* agent-based SIR model in Idris. Note that in the SIR model an agent is in the end just a state-machine thus the model consists of communicating / interacting state-machines. In the book [**?**] the author discusses using dependent types for implementing type-safe state-machines, so we investigate if and how we can apply this to our model. We face the following questions: how can we be total? can we even be total when drawing random-numbers? Also a fundamental question we need to solve then is how we represent time: can we get both the time-semantics of the FRP approach of Haskell AND the type-dependent expressivity or will there be a trade-off between the two?

– TODO: express in the types – SUSCEPTIBLE: MAY become infected when making contact with another agent – INFECTED: WILL recover after a finite number of time-steps – RECOVERED: STAYS recovered all the time

– SIMULATION: advanced in steps, time represented as Nat, as real numbers are not constructive and we want to be total – terminates when there are no more INFECTED agents

show formally that abs does resemble the sd approach: need an idea of a proof and then implement it in dependent types: look at 3 agent system: 2 susceptible, 1 infected. or maybe 2 agents only

Received March 2018; revised March 2018; accepted March 2018