ARTICLE TEMPLATE

Specification Testing of Agent-Based Simulation using Property-Based Testing.

Jonathan Thaler [a] and Peer-Olaf Siebers[a]

[a]School Of Computer Science, University of Nottingham, 7301 Wollaton Rd, Nottingham, UK;

ARTICLE HISTORY
Compiled August 12, 2019

ABSTRACT
This paper explores how to use random property-based testing on a technical level to encode and test specifications of agent-based simulations (ABS). As use case the simple agent-based SIR model is used, where it is shown how to test the complete agent behaviour including its transition probabilities and invariants. The outcome are specifications expressed directly in code, which relate whole classes of random input to expected classes of output. During test execution, random test data is generated automatically, potentially covering the equivalent of thousands of unit tests, run within seconds. This makes property-based testing in the context of ABS strictly more powerful than unit testing, as it is a much more natural fit due to its stochastic nature. The expressiveness and power of property-based testing is not limited to be part of a test-driven development process where it acts as a method for specification, verification and regression tests but can be integrated as a fundamental part of the model development process, supporting hypothesis and discovery making processes. By incorporating this powerful technique into the simulation development process, confidence in the correctness of an implementation is likely to increase dramatically, something of fundamental importance for ABS in general and for ABS supporting far-reaching policy decisions in particular.

KEYWORDS
Agent-Based Simulation Testing; Code Testing; Test Driven Development; Model Specification;

1. Introduction

When implementing an agent-based simulation (ABS) it is of fundamental importance that the implementation is correct up to some specification and that this specification matches the real world in some way. This process is called verification and validation (V&V), where *validation* is the process of ensuring that a model or specification is sufficiently accurate for the purpose at hand whereas *verification* is the process of ensuring that the model design has been transformed into a computer model with sufficient accuracy (Robinson, 2014). In other words, validation determines if we are we building the *right model*, and verification if we are building the *model right* up to some specification (Balci, 1998).

The work of Collier and Ozik (2013) was the first to discuss how to do verification of an ABS implementation, using unit testing with the RePast Framework (North et al.,

CONTACT Jonathan Thaler. Email: jonathan.thaler@nottingham.ac.uk

2013), to verify the correctness of an implementation up to a certain level. Unit testing is a technique, where additional code is written to test specific parts of the implementation. Each test case is constructed manually and expectations about invariants are encoded into assertions. A different approach to testing ABS implementations was investigated by the rather conceptual paper of Thaler and Siebers (2019). In this work the authors introduced *property-based testing* to ABS and showed that it allows to do both verification and validation of an implementation on the code level. The main idea of property-based testing is to express model specifications and laws directly in code and test them through *automated* and *randomised* test data generation. The authors showed that due to ABS' *stochastic*, *exploratory*, *generative* and *constructive* nature, property-based testing is a much more natural fit for testing both explanatory and exploratory ABS than unit testing.

This paper picks up the conceptual work of Thaler and Siebers (2019), puts it into a much more technical perspective and demonstrates additional techniques of property-based testing in the context of ABS, which was not covered in the conceptual paper. More specifically, this paper shows how to encode a full agent specification into property-based tests, using an agent-based SIR model inspired by Macal (2010) as use case. Following an event-driven approach it is shown how to express an agent specification in code by relating random input events to specific output events. Further, showing the use of specific property-based testing features which allow expressing expected coverage of data distributions, it is shown how transition probabilities can be tested. By doing this, this paper demonstrates how property-based testing works on a technical level, how complete specifications can be put into code and how probabilities can be expressed and tested using statistically robust verification. This underlines the result of Thaler and Siebers (2019), that property-based testing maps naturally to ABS. Further, this work shows that in the context of ABS, property-based testing is strictly more powerful than unit testing as it allows to run thousands of test cases automatically instead of constructing each manually and because it is able to encode probabilities, something unit testing is not capable of in general.

The paper is structured as follows: in section 2 property-based testing is introduced on a technical level. In section 3 the agent-based SIR model is introduced, together with its informal event-driven specification. Section 4 is the heart of the paper, where it is shown how to encode agent specifications and transition probabilities with property-based testing. In section 5 the approach is discussed and related to the work of Thaler and Siebers (2019) and other use cases. Finally, section 6 concludes and points out further research.

2. Property-based testing

Property-based testing has its origins in the pure functional programming language Haskell Claessen and Hughes (2000, 2002), where it was first conceived and implemented and has been successfully used for testing Haskell code in the industry for years Hughes (2007).

Property-based testing allows to formulate *functional specifications* in code which then a property-based testing library tries to falsify by *automatically* generating test data, covering as much cases as possible. When a case is found for which the property fails, the library then reduces the test data to its simplest form for which the test still fails, for example shrinking a list to a smaller size. It is clear to see that this kind of testing is especially suited to ABS, because we can formulate specifications, meaning

we describe *what* to test instead of *how* to test. Also the deductive nature of falsification in property-based testing suits very well the constructive and exploratory nature of ABS. Further, the automatic test generation can make testing of large scenarios in ABS feasible because it does not require the programmer to specify all test cases by hand, as is required in traditional unit tests.

Property-based testing was introduced in Claessen and Hughes (2000, 2002) where the authors present the QuickCheck library in Haskell, which tries to falsify the specifications by *randomly* sampling the test space. According to the authors of QuickCheck *"The major limitation is that there is no measurement of test coverage."* Claessen and Hughes (2000). Although QuickCheck provides help to report the distribution of test cases it is not able to measure the coverage of tests in general. This could lead to the case that test cases which would fail are never tested because of the stochastic nature of QuickCheck. Fortunately, the library provides mechanisms for the developer to measure coverage in specific test cases where the data and its expected distribution is known to the developer.

To give a good understanding of how property-based testing works with QuickCheck, we give a few examples of property tests on lists, which are directly expressed as functions in Haskell. Such a function has to return a `Bool` which indicates `True` in case the test succeeds or `False` if not and can take input arguments which data is automatically generated by QuickCheck.

```
-- append operator (++) is associative
append_associative :: [Int] -> [Int] -> [Int] -> Bool
append_associative xs ys zs = (xs ++ ys) ++ zs == xs ++ (ys ++ zs)

-- The reverse of a reversed list is the original list
reverse_reverse :: [Int] -> Bool
reverse_reverse xs = reverse (reverse xs) == xs

-- reverse is distributive over append (++)
reverse_distributive :: [Int] -> [Int] -> Bool
reverse_distributive xs ys = reverse (xs ++ ys) == reverse xs ++ reverse ys
```

When running the tests, we get the following output:

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
*** Failed! Falsifiable (after 5 tests and 6 shrinks):
[0]
[1]
```

We see that QuickCheck generates 100 test cases for each property test and it does this by generating random data for the input arguments. We have not specified any data for our input arguments because QuickCheck is able to provide a suitable data generator through type inference. For lists and all the existing Haskell types there exist custom data generators already. We have to use a monomorphic list, in our case `Int`, and cannot use polymorphic lists because QuickCheck would not know how to generate data for a polymorphic type. Still, by appealing to genericity and polymorphism, we get the guarantee that the test case is the same for all types of a lists.

QuickCheck generates 100 test cases by default and requires all of them to pass. If there is a test case which fails, the overall property test fails and QuickCheck shrinks the input to a minimal size, which still fails and reports it as a counter example. This is the case in the last property test `reverse_distributive` which is wrong as xs and

Figure 1. States and transitions in the SIR compartment model.

ys need to be swapped on the right-hand side. In this run, QuickCheck found a counter example to the property after 5 tests and applied 6 shrinks to find the minimal failing example of `xs = [0]` and `ys = [1]`. If we swap `xs` and `ys`, the property test passes 100 test cases just like the other two did. It is possible to configure QuickCheck to generate more or less random test cases, which can be used to increase the coverage if the sampling space is quite large - this will become useful later.

3. An event-driven agent-based SIR model

The explanatory SIR model is a very well studied and understood compartment model from epidemiology **?**, which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population. The reason for choosing this model is its simplicity as it is easy to understand fully but complex enough to develop basic concepts of pure functional ABS, which are then extended and deepened in the much more complex Sugarscape model of the next section.

In this model, people in a population of size N can be in either one of the 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 reinfection, thus these interactions are ignored in this model. This definition gives rise to three compartments with the transitions seen in Figure 1.

In this paper we want to implement an agent-based simulation of this model, where we follow **??**, translating the informal specification into an an event-driven agent-based approach.

We start by giving the full *specification* of the susceptible, infected and recovered agent by stating the input-to-output event relations. The susceptible agent is specified as follows:

TODO is there some diagram form (BPNL or other process language, e.g. UML), with which we can express the SIR agents event behaviour? would be more concise than only describing it in word

(1) `MakeContact` - if the agent receives this event it will output β `Contact ai Susceptible` events, where `ai` is the agents own id. The events have to be scheduled immediately without delay, thus having the current time as scheduling timestamp. The receivers of the events are uniformly randomly chosen from the agent population. The agent doesn't change its state, stays `Susceptible` and does not schedule any other events than the ones mentioned.
(2) `Contact _ Infected` - if the agent receives this event there is a chance of uniform probability γ (infectivity) that the agent becomes `Infected`. If this happens,

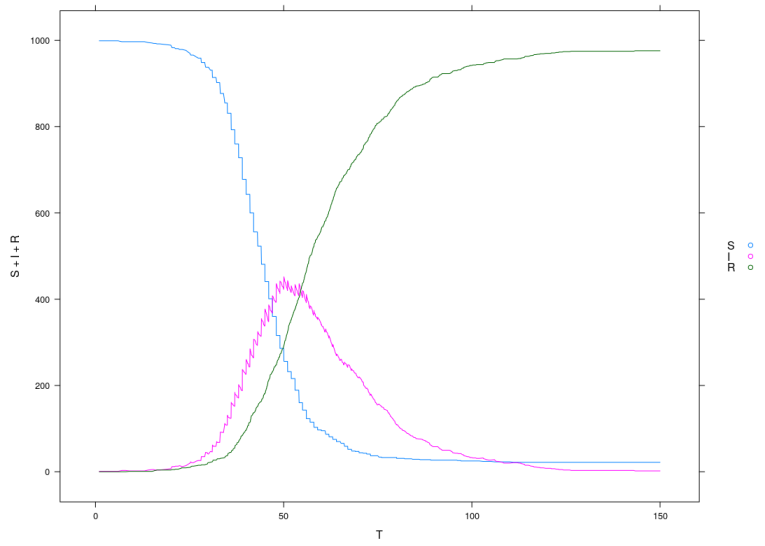

Figure 2. Dynamics of the SIR compartment model using an event-driven agent-based approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent.

the agent will schedule a `Recover` event to itself into the future, where the time is drawn randomly from the exponential distribution with $\lambda = \delta$ (illness duration). If the agent does not become infected, it will not change its state, stays `Susceptible` and does not schedule any events.

(3) `Contact _ _` or `Recover` - if the agent receives any of these other events it will not change its state, stays `Susceptible` and does not schedule any events.

This specification implicitly covers that a susceptible agent can never transition from a `Susceptible` to a `Recovered` state within a single event as it can only make the transition to `Infected` or stay `Susceptible`. The infected agent is specified as follows:

(1) `Recover` - if the agent receives this, it will not schedule any events and make the transition to the `Recovered` state.

(2) `Contact sender Susceptible` - if the agent receives this, it will reply immediately with `Contact ai Infected` to *sender*, where `ai` is the infected agents' id and the scheduling timestamp is the current time. It will not schedule any events and stays `Infected`.

(3) In case of any other event, the agent will not schedule any events and stays `Infected`.

This specification implicitly covers that an infected agent never goes back to the `Susceptible` state as it can only make the transition to `Recovered` or stay `Infected`. From the specification of the susceptible agent it becomes clear that a susceptible agent who became infected, will always recover as the transition to `Infected` includes the scheduling of `Recovered` to itself.

The *recovered* agent specification is very simple. It stays `Recovered` forever and does not schedule any events.

4. Testing agent specifications

In this section we show how to encode the full agent specification given in section 3 in Haskell code and write property tests for it. We conducted our research in Haskell as it is the language where property-based testing has its origins. Obviously we are aware that Haskell is not a mainstream programming language, so to make this paper sufficiently self contained, we introduce concepts step-by-step, which should allow readers, familiar with programming in general, understand the ideas behind what we are doing. Fortunately it is not necessary to go into detail of how agents are implemented as for our approach it is enough to understand the agents' inputs and outputs. For readers interested in the details of how to implement agents in Haskell, we refer to the work of Thaler, Altenkirch, and Siebers (2018). We first start by defining the states of the agents in the SIR model:

```haskell
-- enum of the states the agents can be in
data SIRState = Susceptible | Infected | Recovered
```

Now we define the events as specified in section 3:

```haskell
-- agents are identified by a unique integer
type AgentId = Int
-- enum of the events introduce in specification
data SIREvent
  = MakeContact
  | Contact AgentId SIRState
  | Recover
```

Next, we define a new type for the events to be scheduled, which we termed `QueueItem` as it is put into the event queue. It contains the event to be scheduled, the id of the receiving agent and the scheduling time.

```haskell
type Time      = Double
data QueueItem = QueueItem SIREvent AgentId Time
```

Finally, we define an agent. It is a function mapping an event to the current state of the agent with a list of scheduled events:

```haskell
-- an agent maps an incoming event to the agents current state and a list of scheduled events
sirAgent :: SIREvent -> (SIRState, [QueueItem])
```

We are now ready to discuss how to encode the invariants of the specification from the previous section.

4.1. Encoding the invariants

We start by encoding the invariants of the susceptible agent directly into Haskell, implementing a function which takes all necessary parameters and returns a `Bool` indicating whether the invariants hold or not. The encoding is straightforward when using pattern matching and it nearly reads like a formal specification due to the declarative nature of functional programming.

```haskell
susceptibleProps :: SIREvent              -- ^ Random event sent to agent
                 -> SIRState              -- ^ Output state of the agent
                 -> [QueueItem SIREvent]  -- ^ Events the agent scheduled
                 -> AgentId               -- ^ Agent id of the agent
                 -> Bool
-- received Recover => stay Susceptible, no event scheduled
```

```
susceptibleProps Recover Susceptible es _ = null es
-- received MakeContact => stay Susceptible, check events
susceptibleProps MakeContact Susceptible es ai
  = checkMakeContactInvariants ai es cor
-- received Contact _ Recovered => stay Susceptible, no event scheduled
susceptibleProps (Contact _ Recovered) Susceptible es _ = null es
-- received Contact _ Susceptible => stay Susceptible, no event scheduled
susceptibleProps (Contact _ Susceptible) Susceptible es _  = null es
-- received Contact _ Infected, didn't get Infected, no event scheduled
susceptibleProps (Contact _ Infected) Susceptible es _ = null es
-- received Contact _ Infected AND got infected, check events
susceptibleProps (Contact _ Infected) Infected es ai
  = checkInfectedInvariants ai es
-- all other cases are invalid and result in a failed test case
susceptibleProps _ _ _ _ = False
```

Next, we give the implementation for the `checkMakeContactInvariants` function. We omit a detailed implementation of `checkInfectedInvariants` as it works in a similar way and its details do not add anything conceptually new. The function `checkMakeContactInvariants` encodes the invariants which have to hold when the susceptible agent receives a `MakeContact` event:

```
checkInfectedInvariants :: AgentId              -- ^ Agent id of the agent
                        -> [QueueItem SIREvent] -- ^ Events the agent scheduled
                        -> Bool
checkInfectedInvariants sender
  -- expect exactly one Recovery event
  [QueueItem receiver (Event Recover) t']
  -- receiver is sender (self) and scheduled into the future
  = sender == receiver && t' >= t
-- all other cases are invalid
checkInfectedInvariants _ _ = False
```

4.2. Writing a property test

What is left is to actually write a property test using QuickCheck. We are making heavy use of random parameters to express that the properties have to hold invariant of the model parameters. We make use of additional data generator modifiers: `Positive` ensures that the value generated is positive; `NonEmptyList` ensures that the randomly generated list is not empty.

QuickCheck comes with a lot of data generators for existing types like `String`, `Int`, `Double`, `[]`, but in case one wants to randomize custom data types one has to write custom data generators. There are two ways to do this. Either fix them at compile time by writing an `Arbitrary` instance or write a run-time generator running in the `Gen` context. The advantage of having an `Arbitrary` instance is that the custom data type can then be used as random argument to a function..

This implementation makes use of the `elements :: [a]` $\rightarrow$ `Gen a` functions, which picks a random element from a non-empty list with uniform probability. If a skewed distribution is needed, one can use the `frequency :: [(Int, Gen a)]` $\rightarrow$ `Gen a` function, where a frequency can be specified for each element.

```
genEventFreq :: Int
             -> Int
             -> Int
             -> (Int, Int, Int)
             -> [AgentId]
             -> Gen SIREvent
genEventFreq mcf _ rcf _ []
```

```
  = frequency [ (mcf, return MakeContact), (rcf, return Recover)]
genEventFreq mcf cof rcf (s,i,r) ais
  = frequency [ (mcf, return MakeContact)
              , (cof, do
                  ss <- frequency [ (s, return Susceptible)
                                  , (i, return Infected)
                                  , (r, return Recovered)]
                  ai <- elements ais
                  return (Contact ai ss))
              , (rcf, return Recover)]

genEvent :: [AgentId] -> Gen SIREvent
genEvent = genEventFreq 1 1 1 (1,1,1)
```

When we have a random `Double` as input to a function but want to restrict its random range to (0,1) because it reflects a probability, we can do this easily with `newtype` and implementing an `Arbitrary` instance. The same can be done for limiting the simulation duration to a lower range than the full `Double` range. Implementing an `Arbitrary` instance is straightforward, one only needs to implement the `arbitrary :: Gen a` method:

```
newtype Probability = P Double
newtype TimeRange   = T Double

instance Arbitrary Probability where
  arbitrary = P <$> choose (0, 1)

instance Arbitrary TimeRange where
  arbitrary = T <$> choose (0, 50)
```

We are now equipped with all functionality to implement the property test.

```
prop_susceptible_invariants :: Positive Int          -- ^ Contact rate (beta)
                            -> Probability           -- ^ Infectivity (gamma)
                            -> Positive Double       -- ^ Illness duration (delta)
                            -> Positive Double       -- ^ Current simulation time
                            -> NonEmptyList AgentId -- ^ population agent ids
                            -> Gen Property
prop_susceptible_invariants
  (Positive beta) (P gamma) (Positive delta) (Positive t) (NonEmpty ais) = do
  -- generate random event, requires the population agent ids
  evt <- genEvent ais
  -- run susceptible random agent with given parameters
  (ai, ao, es) <- genRunSusceptibleAgent beta gamma delta t ais evt
  -- check properties
  return (label (labelTestCase ao) (property (susceptibleProps evt ao es ai)))
  where
    labelTestCase :: SIRState -> String
    labelTestCase Infected    = "Susceptible -> Infected"
    labelTestCase Susceptible = "Susceptible"
    labelTestCase Recovered   = "INVALID"
```

Due to the large random sampling space with 5 parameters, we increase the number of test cases to generate to 100,000. We also label the test cases to generate a distribution of the transitions. The case where the agents output state is `Recovered` is marked as "INVALID" as it must never occur, otherwise the test will fail, due to the invariants encoded above.

```
+++ OK, passed 100000 tests (6.77s):
94.522% Susceptible
 5.478% Susceptible -> Infected
```

All 100,000 test cases go through within 6.7 seconds. The distribution of the transitions shows that we indeed cover both cases a susceptible agent can react within one event. It either stays susceptible or makes the transition to infection. The fact that there is no transition to recovered shows that the implementation is correct - for a transition to recovered we would need to send an additional, second event to the agent.

Encoding of the invariants and writing property tests for the infected agents follows the same idea and is not repeated here. Next, we show how to test transition probabilities using the powerful statistical hypothesis testing feature of QuickCheck.

4.3. Encoding transition probabilities

In the specifications from section 3 there are probabilistic state transitions, for example an infected agent *will* recover after a given time, which is randomly distributed with the exponential distribution. The susceptible agent *might* become infected, depending on the events it receives and the infectivity (γ) parameter. We look now into how we can encode these probabilistic properties using the powerful `cover` and `checkCoverage` feature of QuickCheck.

The function `cover :: Testable prop ⇒ Double → Bool → String → prop → Property` allows to explicitly specify that a given percentage of successful test cases belong to a given class. The first argument is the expected percentage; the second argument is a `Bool` indicating whether the current test case belongs to the class or not; the third argument is a label for the coverage; the fourth argument is the property which needs to hold for the test case to succeed.

QuickCheck provides the powerful function `checkCoverage :: Testable prop ⇒ prop → Property` which does this for us. When `checkCoverage` is used, QuickCheck will run an increasing number of test cases until it can decide whether the percentage in `cover` was reached or cannot be reached at all. The way QuickCheck does it, is by using sequential statistical hypothesis testing Wald (1992), thus if QuickCheck comes to the conclusion that the given percentage can or cannot be reached, it is based on a robust statistical test giving strong confidence in the result.

We follow the same approach as in encoding the invariants of the susceptible agent but instead of checking the invariants, we compute the probability for each case. In this property test we cannot randomise the model parameters because this would lead to random coverage. This might seem like a disadvantage but we do not really have a choice here, still the model parameters can be adjusted arbitrarily and the property must hold. We make use of the `cover` function together with `checkCoverage`, which ensures that we get a statistical robust estimate whether the expected percentages can be reached or not. Implementing this property test is then simply a matter of computing the probabilities and of case analysis over the random input event and the agents output.

```
...
case evt of
  Recover ->
    cover recoverPerc True
     ("Susceptible receives Recover, expected " ++ show recoverPerc) True
...
```

Note the usage pattern of `cover` where we unconditionally include the test case into the coverage class so all test cases pass. The reason for this is that we are just interested in testing the coverage, which is in fact the property we want to test. We

could have combined this test into the previous one but then we couldn't have used randomised model parameters. For this reason, and to keep the concerns separated we opted for two different tests, which makes them also much more readable.

When running the property test we get the following output:

```
+++ OK, passed 819200 tests (7.32s):
33.3582% Susceptible receives MakeContact, expected 33.33%
33.2578% Susceptible receives Recover, expected 33.33%
11.1643% Susceptible receives Contact * Recovered, expected 11.11%
11.1096% Susceptible receives Contact * Susceptible, expected 11.11%
10.5616% Susceptible receives Contact * Infected, stays Susceptible, expected 10.56%
 0.5485% Susceptible receives Contact * Infected, becomes Infected, expected 0.56%
```

After 819,200 (!) test cases QuickCheck comes to the conclusion that the distributions generated by the test cases reflect the expected distributions and passes the property test. We see that the values do not match exactly in some cases but by using sequential statistical hypothesis testing, QuickCheck is able to conclude that the coverage are statistically equal.

5. Discussion

TODO: statistical sequential hypothesis testing can also be applied to exploratory models like the sugarscape as shown in the conceptual paper, comparing two different implementations of the same model for example compare the distributions of a time- and event-driven implementation, encode model invariants

5.1. Emulating failure

As already mentioned, *all* test cases have to pass for the whole property test to succeed. If just a single test case fails, the whole property test fails. This requirement is sometimes too strong, especially when we are dealing with stochastic systems like ABS.

The function `cover` can be used to emulate failure of test cases and get a measure of failure. Instead of computing the `True/False` property in the last `prop` argument, we set the last argument always to `True` and compute the `True/False` property in the second `Bool` argument, indicating whether the test case belongs to the class of passed tests or not. This has the effect that *all* test cases are successful but that we get a distribution of failed and successful ones. In combination with `checkCoverage`, this is a particularly powerful pattern for testing ABS, which allows us to test hypotheses and statistical tests on distributions as will be shown in the following chapters.

6. Conclusions

hypothesise that a strong reason for why testing in ABS is not very widely used and adopted is that unit testing is not able to deal very well with the stochastic nature of ABS in general. random property-based testing is a remedy to that problem as it allows to relate whole classes of inputs to specific classes of output for which then randomised test cases are automatically generated, covering potentially thousands of unit tests.

benefits: - express specifications rather than individual test cases which makes it much more general than unit testing - expressing probabilities of various types (hypotheses, transitions, outputs) and perform statistical robust testing by sequential hypothesis testing - relates whole classes of inputs to whole classes of outputs, automatically generating thousands of tests if necessary,

drawbacks: - coverage but smallcheck could be a remedy As a remedy for the potential coverage problems of QuickCheck, there exists also a deterministic property-testing library called SmallCheck Runciman, Naylor, and Lindblad (2008) which instead of randomly sampling the test space, enumerates test cases exhaustively up to some depth

6.1. Further Research

The transitions we implemented were only one-step transitions, feeding only a single event. Although we covered the full functionality by also testing the infected and recovered agent separately (which was not shown in the paper due to space limitations), we could also implement property tests which test the full transition from susceptible to recovered, which then would required multiple events and slightly different approach calculating the probabilities.

References

Balci, O. (1998). Verification, Validation, and Testing. In J. Banks (Ed.), *Handbook of Simulation* (pp. 335–393). John Wiley & Sons, Inc. Retrieved 2017-05-18, from `http://onlinelibrary.wiley.com/doi/10.1002/9780470172445.ch10/summary`

Claessen, K., & Hughes, J. (2000). QuickCheck - A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (pp. 268–279). New York, NY, USA: ACM. Retrieved 2016-11-16, from `http://doi.acm.org/10.1145/351240.351266`

Claessen, K., & Hughes, J. (2002, December). Testing Monadic Code with QuickCheck. *SIGPLAN Not.*, *37*(12), 47–59. Retrieved 2017-05-11, from `http://doi.acm.org/10.1145/636517.636527`

Collier, N., & Ozik, J. (2013, December). Test-driven agent-based simulation development. In *2013 Winter Simulations Conference (WSC)* (pp. 1551–1559).

Hughes, J. (2007). QuickCheck Testing for Fun and Profit. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages* (pp. 1–32). Berlin, Heidelberg: Springer-Verlag. Retrieved 2018-09-24, from `http://dx.doi.org/10.1007/978-3-540-69611-7`$_1$

Macal, C. M. (2010). To Agent-based Simulation from System Dynamics. In *Proceedings of the Winter Simulation Conference* (pp. 371–382). Baltimore, Maryland: Winter Simulation Conference. Retrieved 2017-10-05, from `http://dl.acm.org/citation.cfm?id=2433508.2433551`

North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M., & Sydelko, P. (2013, March). Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling*, *1*(1), 3. Retrieved 2018-08-02, from `https://doi.org/10.1186/2194-3206-1-3`

Robinson, S. (2014). *Simulation: The Practice of Model Development and Use*. Macmillan Education UK. (Google-Books-ID: Dtn0oAEACAAJ)

Runciman, C., Naylor, M., & Lindblad, F. (2008). Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (pp. 37–48). New York, NY, USA: ACM. Retrieved 2018-09-18, from `http://doi.acm.org/10.1145/1411286.1411292`

Thaler, J., Altenkirch, T., & Siebers, P.-O. (2018). Pure Functional Epidemics: An Agent-Based Approach. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages* (pp. 1–12). New York, NY, USA: ACM. Retrieved 2019-05-03, from `http://doi.acm.org/10.1145/3310232.3310372` (event-place: Lowell, MA, USA)

Thaler, J., & Siebers, P.-O. (2019, July). Show Me Your Properties! The Potential Of Property-Based Testing In Agent-Based Simulation. Berlin.

Wald, A. (1992). Sequential Tests of Statistical Hypotheses. In S. Kotz & N. L. Johnson (Eds.), *Breakthroughs in Statistics: Foundations and Basic Theory* (pp. 256–298). New York, NY: Springer New York. Retrieved 2019-05-03, from `https://doi.org/10.1007/978-1-4612-0919-5`$_1 8$