Autonomous Agents and Multi-Agent Systems manuscript No.
(will be inserted by the editor)

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

Jonathan Thaler · Peer-Olaf Siebers

Received: TODO / Accepted: TODO

Abstract This paper explores how to use 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 agent behaviour including 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 models supporting far-reaching policy decision in particular.

Keywords Agent-Based Simulation Testing · Code Testing · Test Driven Development · Model Specification

Both Authors
School Of Computer Science
University of Nottingham
7301 Wollaton Rd, Nottingham, UK
E-mail: jonathan.thaler@nottingham.ac.uk

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 [9]. 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 [1].

The work of [4] was the first to discuss how to do verification of an ABS implementation, using unit testing with the RePast Framework [8], 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 [13]. 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 [13], 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 agent specifications into property-based tests, using an agent-based SIR model inspired by [7] 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 specifications can be put into code and how probabilities can be expressed and tested using statistically robust verification. This underlines the conclusion of [13], 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 [13] and other use cases. Finally, section 6 concludes and points out further research.

2 Property-based testing

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 necessary. 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 it allows to formulate specifications, where we describe *what* to test instead of *how* to test. 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 has its origins in the pure functional programming language Haskell in the works of [2,3], where the authors present the QuickCheck library, which tries to falsify the specifications by *randomly* sampling the test space. According to [5], this library has been successfully used for testing Haskell code in the industry for years , underlining its maturity and real world relevance in general and of property-based testing in particular.

To give a good understanding of how property-based testing works with QuickCheck, we give examples of properties of lists, which are directly expressed as functions in Haskell. Such functions can take arbitrary inputs, which random data are generated automatically by QuickCheck during testing. The return type of the function is a `Bool` indicating whether the property holds for the given random inputs or not.

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

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

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

When testing each property with QuickCheck, we get the following output:

```
> quickCheck prop_append_associative
+++ OK, passed 100 tests.
> quickCheck prop_reverse_reverse
```

```
+++ OK, passed 100 tests.
> quickCheck prop_reverse_distributive
*** 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 as used in these examples and all the existing Haskell types there exist custom data generators already.

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 third property test `prop_reverse_distributive` which is wrong as `xs` and `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 an arbitrary number of random test cases, which can be used to increase the coverage if the sampling space is quite large.

3 Event-driven agent-based SIR model

As use case to develop the concepts in this paper, we use the explanatory SIR model by [6]. It 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.

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 follow [7] for translating the informal SIR specification into an event-driven agent-based approach. The dynamics it produces are shown in Figure 2, which was generated by our own implementation undertaken for this paper, accessible in our repository [11].

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

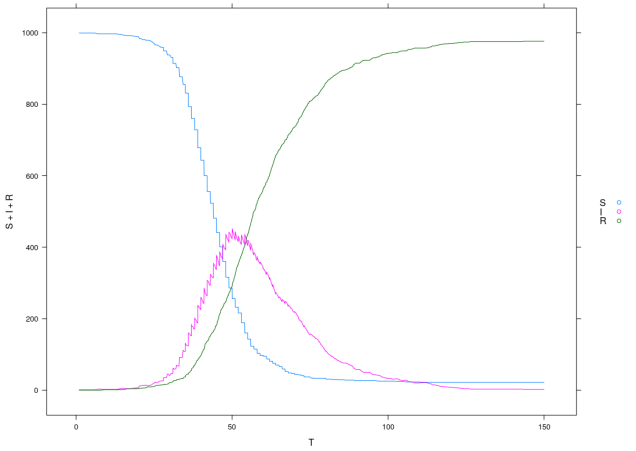

Fig. 2 Dynamics of the SIR compartment model using an event-driven agent-based approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent.

3.1 An informal specification

In this section we give an informal specification of the agent behaviour, relating the input to according output events. Before we can do that we first need to define the event types of the model, how they related to scheduling and how we can conceptually represent agents.

We are using Haskell as notation and implementation as we conducted our research in that language because it originated property-based testing. 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 ABS in Haskell, we refer to the work of [12].

We start by defining the states the agents can be in:

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

The model uses three types of events. First, `MakeContact` is used by a susceptible agent to proactively make contact with β other agents per time

unit by scheduling it to itself. Second, `Contact` is used by susceptible and infected agents to contact other agents, revealing their id and their state to the receiver. Third, `Recover` is used by an infected agent to proactively make the transition to recovered after δ time units.

```
-- agents are identified by a unique integer
type AgentId = Int
-- enumeration of the three events
data SIREvent = MakeContact | Contact AgentId SIRState | Recover
```

As events are scheduled we need a new type to hold them 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.

```
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. This is a simplified view on how agents are actually implemented in Haskell but it suffices for our purpose.

```
-- 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 give 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:

1. `MakeContact` - If the agent receives this event it will output β `(Contact ai Susceptible)` events, where `ai` is the agents own id and `Susceptible` indicating the event comes from a susceptible agent. 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 that the agent becomes `Infected`. If this happens, the agent will schedule a `Recover` event to itself into the future, where the time is drawn randomly from the exponential distribution with $\lambda = \delta$. 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 but 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 Encoding the agent specification

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.

```
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 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 MakeContact => stay Susceptible, check events
susceptibleProps MakeContact Susceptible es ai
  = checkMakeContactInvariants ai es cor
-- 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
```

4.1 Writing a property test

After having encoded the invariants into a function, we need to write a QuickCheck property test which calls this function with random test data. Although QuickCheck comes with a lot of data generators for existing types like Strings, Integers, Double, List, it obviously does not have generators for custom types, like the `SIRState` and `SIREvent`. Thus, the first step is to write custom data generators, which are suitable for our problem at hand.

There are two ways to do this by either fixing them at compile time writing an `Arbitrary` instance or writing 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 an argument to a function, the advantage of writing a run-time generator is that it can depend on values computed during run time.

First, we write an `Arbitrary` instance for the `SIRState`. When writing an `Arbitrary` instance one needs to provide an implementation for the `arbitrary` method, which returns a value of the given type for which the instance was implemented for - in our case it is `SIRState`. This implementation makes use of the `elements` function, which picks a random element from a non-empty list with uniform probability.

```
instance Arbitrary SIRState where
  arbitrary = elements [Susceptible, Infected, Recovered]
```

Next we write a custom generator to capture the concept of probabilities. This is necessary because if using a `Double` as input to a function it would cover the whole range of the type but we want to restrict its random range to $(0, 1)$ reflecting a probability. We achieve this by introducing a new type and writing an `Arbitrary` instance for it. This implementation makes use of the `choose` function, which returns a random value within the given bounds.

```
-- define a new type, representing a probability
newtype Probability = P Double
instance Arbitrary Probability where
  arbitrary = P <$> choose (0, 1)
```

What is left is to write a custom generator for generating random events. In this case we want to be able to control the distribution of events generated. Instead of using `elements`, which picks uniformly, if a skewed distribution is needed, one can use the `frequency :: [(Int, Gen a)]` → `Gen a` function, where a frequency can be specified for each element. Due to the fact, that event generation is parametrised by frequencies and requires the population ids to pick from in the `Contact` event, it needs to be a run-time generator, thus running in `Gen` context. The function takes the frequencies for all three types of events, the frequencies of the `SIRState` in the `Contact` event and the agent population to pick the sender id from in the `Contact` event.

```
genEventFreq :: Int             -- MakeContact frequency
             -> Int             -- Recover frequency
             -> Int             -- Contact frequency
             -> (Int, Int, Int) -- Susceptible, Infected, Recovered frequency
             -> [AgentId]       -- Population ids
             -> Gen SIREvent
genEventFreq mf cf rf (s,i,r) ais
  -- generate one of the three events with given frequency
  = frequency [ (mf, return MakeContact)
              , (cf, do
                  -- pick SIRState for Contact event
                  ss <- frequency [ (s, return Susceptible)
                                  , (i, return Infected)
                                  , (r, return Recovered)]
                  -- pick sender id for Contact event
                  ai <- elements ais
                  return (Contact ai ss))
              , (rf, return Recover)]

-- helper function with uniform frequencies for all events and SIRState
genEvent :: [AgentId] -> Gen SIREvent
genEvent = genEventFreq 1 1 1 (1,1,1)
```

We are now equipped with all functionality to implement the property test. All parameters to the property test are generated randomly, which expresses that the properties encoded in the previous section have to hold invariant of the model parameters. We make use of additional data generator modifiers: `Positive` ensures that a value generated is positive; `NonEmptyList` ensures that a randomly generated list is not empty. Further, we use `label` which can be used to generate labels for a test case, counting occurrences of same labels, generating a histogram, which is useful to get an insight into the distribution of the property test data. We use it to get an understanding for the 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 in the previous section.

```
prop_susceptible :: Positive Int         -- ^ Contact rate (beta)
                 -> Probability          -- ^ Infectivity (gamma)
                 -> Positive Double      -- ^ Illness duration (delta)
                 -> Positive Double      -- ^ current simulation time
                 -> NonEmptyList AgentId -- ^ population agent ids
                 -> Gen Bool
```

```
prop_susceptible
  (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 (implementation omitted)
  (ai, ao, es) <- genRunSusceptibleAgent beta gamma delta t ais evt
  -- check properties
  return (label (labelTestCase ao) (susceptibleProps evt ao es ai))
  where
    labelTestCase :: SIRState -> String
    labelTestCase Infected    = "Susceptible -> Infected"
    labelTestCase Susceptible = "Susceptible"
    labelTestCase Recovered   = "INVALID"
```

We have omitted the implementation of `genRunSusceptibleAgent` as it would require the discussion of implementation details of the agent. Conceptually speaking, it executes the agent with the respective arguments with a fresh random-number generator and returns the agent id, its state and scheduled events it has output.

Finally we can run the test using QuickCheck. Due to the large random sampling space with 5 parameters, we increase the number of test cases to generate to 100,000.

```
> quickCheckWith (stdArgs {maxSuccess=100000}) prop_susceptible
+++ OK, passed 100000 tests (6.77s):
94.522% Susceptible
 5.478% Susceptible -> Infected
```

All 100,000 test cases pass, taking 6.7 seconds to run on our machine. The distribution of the transitions shows that we indeed cover both cases a susceptible agent can exhibit 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.

4.2 Encoding transition probabilities

In the specifications from the previous section there are probabilistic state transitions, for example 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` feature of QuickCheck.

The function `cover :: 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 indicates 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.

An example would be to use it to check whether at least 15% of all test cases of the `prop_reverse_reverse` property of section 2 has a length of at least 50. This would be encoded in the following way:

```
prop_reverse_reverse_cover :: [Int] -> Property
prop_reverse_reverse_cover xs  =
  cover 15 (length xs >= 50) "length at least 50" (reverse (reverse xs) == xs)
```

For our case we follow a slightly different approach: we include all test cases into the expected coverage, setting the second parameter always to `True` and we also set the last argument to `True` as we are only interested in testing the coverage, which is in fact the property we want to test. 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. It is important to note that in this property test we cannot randomise the model parameters β, γ and δ 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 still hold.

```
prop_susceptible_proab :: Positive Double      -- Current simulation time
                       -> NonEmptyList AgentId -- Agent ids of the population
                       -> Property
prop_susceptible_proab (Positive t) (NonEmpty ais) = do
  -- fixed model parameters, otherwise random coverage
  let cor = 5     -- contact rate (beta)
      inf = 0.05  -- infectivity (gamma)
      ild = 15.0  -- illness duration (delta)
  -- compute distributions for all cases depending on event and SIRState
  -- frequencies; technical detail, omitted for clarity reasons
  let recoverPerc       = ...
      makeContPerc      = ...
      contactRecPerc    = ...
      contactSusPerc    = ...
      contactInfSusPerc = ...
      contactInfInfPerc = ...
  -- generate a random event
  evt <- genEvent ais
  -- run susceptible random agent with given parameters
  (_, ao, _) <- genRunSusceptibleAgent cor inf ild t ais evt
  -- encode expected distributions
  return $ property $ case evt of
    Recover ->
      cover recoverPerc True "Susceptible recv Recover" True
    MakeContact ->
      cover makeContPerc True "Susceptible recv MakeContact" True
    (Contact _ Recovered) ->
      cover contactRecPerc True "Susceptible recv Contact * Recovered" True
    (Contact _ Susceptible) ->
      cover contactSusPerc True "Susceptible recv Contact * Susceptible" True
    (Contact _ Infected) ->
      case ao of
        Susceptible ->
```

```
      cover contactInfSusPerc True
        "Susceptible recv Contact * Infected, stays Susceptible" True
    Infected ->
      cover contactInfInfPerc True
        "Susceptible recv Contact * Infected, becomes Infected" True
    _ ->
      cover 0 True "INVALID" True
```

The usage pattern of `cover`, where we unconditionally include the test case into the coverage class so all test cases pass, emulates failure. 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.

We have omitted the details of computing the respective distributions of the cases, which depend on the frequencies of the events and the occurrences of `SIRState` within the `Contact` event. By varying different distributions in the `genEvent` function, we can change the distribution of the test cases, leading to a more general test than just using uniform distributed events. When running the property test we get the following output:

```
+++ OK, passed 100 tests (0.01s):
40% Susceptible recv MakeContact
25% Susceptible recv Recover
14% Susceptible recv Contact * Infected, stays Susceptible
12% Susceptible recv Contact * Susceptible
 9% Susceptible recv Contact * Recovered

Only 9% Susceptible recv Contact * Recovered, but expected 11%
Only 25% Susceptible recv Recover, but expected 33%
```

QuickCheck runs 100 test cases, prints the distribution of our labels and issues warnings in the last two lines that generated and expected coverages differ in these cases. Further, not all cases are covered, for example the contact with an infected and becoming infected. The reason for these issues is insufficient testing coverage as 100 test cases are simply not enough for a statistically robust result. We could increase the number of test cases to run for example to 100,000 which will cover then all cases but still QuickCheck is not satisfied as the expected and generated coverage differs in some fractions.

Fortunately, as a remedy QuickCheck provides the function `checkCoverage :: prop → Property` which solves this problem. When `checkCoverage` is added to the top of the function before the `do`, QuickCheck will run as many test cases necessary until it is clear 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 based on the work of [14]. 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. With the usage of `checkCoverage` we get the following output:

```
+++ OK, passed 819200 tests (7.32s):
33.3292% Susceptible recv Recover
```

```
33.2697% Susceptible recv MakeContact
11.1921% Susceptible recv Contact * Susceptible
11.1213% Susceptible recv Contact * Recovered
10.5356% Susceptible recv Contact * Infected, stays Susceptible
 0.5520% Susceptible recv Contact * Infected, becomes Infected
```

After 819,200 (!) test cases, run in 7.32 seconds on our machine, QuickCheck comes to the statistically robust conclusion that the distributions generated by the test cases reflect the expected distributions and passes the property test.

5 Discussion

We have shown how to encode the *informal* specification of the susceptible agent behaviour into a *formal* one through the use of functions and property tests. We have omitted tests for the infected agents as they follow conceptually the same patterns. The testing of transitions of the infected agents work slightly different though as they follow an exponential distribution but are encoded in an equal fashion as demonstrated with the susceptible agent. The case for the recovered agent is a bit more subtle, due to its behaviour: it simply stays `Recovered` *forever*. This is a property which cannot be tested in a reasonable way with neither property-based nor unit testing as it is simply not computable. Only a white box test in the case of a code review would reveal whether the implementation is actually correct or not.

Statistical sequential hypothesis testing can also be applied to the example of hypothesis testing in exploratory models in the paper of [13]. In their case the expected coverage is encoded as in our case but instead of passing all tests unconditionally, a property is checked and the outcome is passed as the last argument to `cover`.

Another useful application for `cover` and `checkCoverage` are checking whether two different implementations of the same model result in the same distributions *under random model parameters*. Another use case is encoding model invariants of the distribution generated, for example in the case of the SIR model, the number of agents stays always constant, the number of susceptible is monotonic decreasing and the number of recovered monotonic decreasing *under random model parameters*.

6 Conclusions

In this paper we have shown how to use property-based testing on a technical level to encode informal specifications of agent behaviour into formal specification directly in code. The benefits of a property-based approach in ABS over unit testing is manifold. First, it expresses specifications rather than individual test cases which makes it more general than unit testing. It allows expressing probabilities of various types (hypotheses, transitions, outputs) and perform statistically robust testing by sequential hypothesis testing. It relates whole

classes of inputs to whole classes of outputs, automatically generating thousands of tests if necessary, ran within seconds, depending on the complexity of the test.

The main drawback is to get the coverage of the tests right, which is not always obvious from the beginning. As a remedy for the potential coverage problems of QuickCheck, there exists also a *deterministic* property-testing library called SmallCheck of [10], 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 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), the next step is to implement property tests which test the full transition from susceptible to recovered, which then would required multiple events and a different approach calculating the probabilities.

References

1. Balci, O.: Verification, Validation, and Testing. In: J. Banks (ed.) Handbook of Simulation, pp. 335–393. John Wiley & Sons, Inc. (1998). DOI 10.1002/9780470172445.ch10. URL http://onlinelibrary.wiley.com/doi/10.1002/9780470172445.ch10/summary
2. Claessen, K., Hughes, J.: QuickCheck - A Lightweight Tool for Random Testing of Haskell Programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, pp. 268–279. ACM, New York, NY, USA (2000). DOI 10.1145/351240.351266. URL http://doi.acm.org/10.1145/351240.351266
3. Claessen, K., Hughes, J.: Testing Monadic Code with QuickCheck. SIGPLAN Not. **37**(12), 47–59 (2002). DOI 10.1145/636517.636527. URL http://doi.acm.org/10.1145/636517.636527
4. Collier, N., Ozik, J.: Test-driven agent-based simulation development. In: 2013 Winter Simulations Conference (WSC), pp. 1551–1559 (2013). DOI 10.1109/WSC.2013.6721538
5. Hughes, J.: QuickCheck Testing for Fun and Profit. In: Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages, PADL'07, pp. 1–32. Springer-Verlag, Berlin, Heidelberg (2007). DOI 10.1007/978-3-540-69611-7_1. URL http://dx.doi.org/10.1007/978-3-540-69611-7_1
6. Kermack, W.O., McKendrick, A.G.: A Contribution to the Mathematical Theory of Epidemics. Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences **115**(772), 700–721 (1927). DOI 10.1098/rspa.1927.0118. URL http://rspa.royalsocietypublishing.org/content/115/772/700
7. Macal, C.M.: To Agent-based Simulation from System Dynamics. In: Proceedings of the Winter Simulation Conference, WSC '10, pp. 371–382. Winter Simulation Conference, Baltimore, Maryland (2010). URL http://dl.acm.org/citation.cfm?id=2433508.2433551
8. North, M.J., Collier, N.T., Ozik, J., Tatara, E.R., Macal, C.M., Bragen, M., Sydelko, P.: Complex adaptive systems modeling with Repast Simphony. Complex Adaptive Systems Modeling **1**(1), 3 (2013). DOI 10.1186/2194-3206-1-3. URL https://doi.org/10.1186/2194-3206-1-3
9. Robinson, S.: Simulation: The Practice of Model Development and Use. Macmillan Education UK (2014). Google-Books-ID: Dtn0oAEACAAJ

10. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell, Haskell '08, pp. 37–48. ACM, New York, NY, USA (2008). DOI 10.1145/1411286.1411292. URL http://doi.acm.org/10.1145/1411286.1411292
11. Thaler, J.: Repository of Agent-Based SIR implementation in Haskell (2019). URL https://github.com/thalerjonathan/haskell-sir
12. Thaler, J., Altenkirch, T., Siebers, P.O.: Pure Functional Epidemics: An Agent-Based Approach. In: Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, IFL 2018, pp. 1–12. ACM, New York, NY, USA (2018). DOI 10.1145/3310232.3310372. URL http://doi.acm.org/10.1145/3310232.3310372. Event-place: Lowell, MA, USA
13. Thaler, J., Siebers, P.O.: Show Me Your Properties! The Potential Of Property-Based Testing In Agent-Based Simulation. Berlin (2019)
14. Wald, A.: Sequential Tests of Statistical Hypotheses. In: S. Kotz, N.L. Johnson (eds.) Breakthroughs in Statistics: Foundations and Basic Theory, Springer Series in Statistics, pp. 256–298. Springer New York, New York, NY (1992). DOI 10.1007/978-1-4612-0919-5_18. URL https://doi.org/10.1007/978-1-4612-0919-5_18