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 July 29, 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). The claim is that opposed to unit testing, random property-based testing is a much more natural fit to test ABS due to both stochastic nature. The paper shows how to test full agent- and model-specifications, in the case of an agents behaviour, its transition probabilities and model 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. The expressiveness and power of property-based testing is not only limited to be part of a test-driven development process where it acts as specification, verification and regression test but can be integrated as a fundamental part of the model development process, supporting the hypothesis and discovery making process. By incorporating this powerful technique into the simulation development process the confidence in the correctness of an implementation increases dramatically, something of fundamental importance for ABS in general and for ABS supporting far-reaching policy decisions in particular.

KEYWORDS
Testing; Test Driven Development; Model Specification;

1. Introduction

When implementing an 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 (14). 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 (2).

The work (6) was the first to discuss how to do verification of an ABS implementation, using unit testing with the RePast Framework (10), to verify the correctness of the implementation up to a certain level. Unit testing is a technique where additional tests are written in code by constructing individual test cases, to test a specific

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

unit of the implementation. A different approach to testing an implementation of an ABS was investigated by the rather conceptual paper of (**?**). 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. Property-based testing has its origins in Haskell (4; 5), where it was first conceived and implemented and has been successfully used for testing Haskell code in the industry for years (9).

This paper picks up the conceptual work of (**?**), 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.

- full agent specification of explanatory SIR model inspired by macal - showing how to encode transitions and probabilities and test them using statistical robust verification, random event sampling

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 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 (4; 5) 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."* (4). 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. This is a powerful tool for testing randomness in ABS as will be shown in the next chapters.

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 (++)
-- This test fails for explanatory reasons, for a correct
-- property xs and ys need to be swapped on the right-hand side!
reverse_distributive :: [Int] -> [Int] -> Bool
reverse_distributive xs ys = reverse (xs ++ ys) == reverse xs ++ reverse ys

-- running the tests
main :: IO ()
main = do
  quickCheck append_associative
  quickCheck reverse_reverse
  quickCheck reverse_distributive
```

When we run the tests using *main*, 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 *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.

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

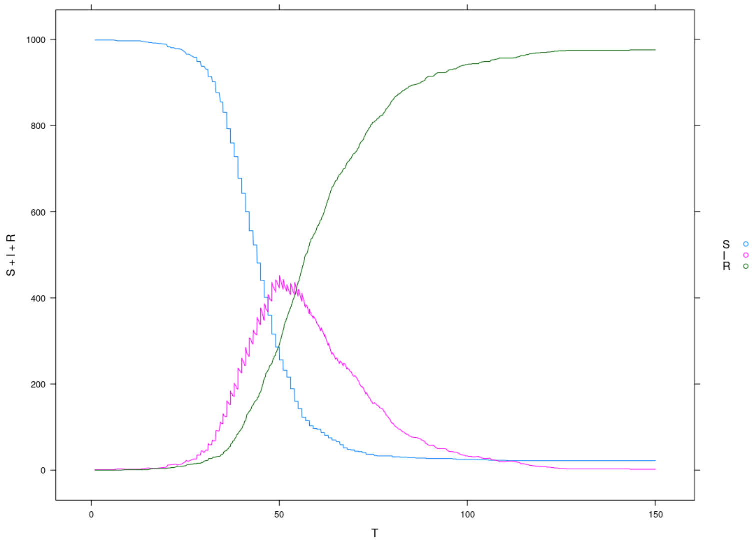

Figure 2. Dynamics of the SIR compartment model using an event-driven agent-based approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent.

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 TODO: cite macal, 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, 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. Implementing agent specifications

TODO: put emphasis on statistical robust testing using cover and checkCoverage, TODO: introduce concepts along writing the main body. focus on event-driven sir. TODO: it would be great if i can show how property-based testing found a bug in an implementation

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
```

```haskell
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` and `checkInfectedInvariants` functions. The function `checkMakeContactInvariants` encodes the invariants which have to hold when the susceptible agent receives a `MakeContact` event. The `checkInfectedInvariants` function encodes the invariants which have to hold when the susceptible agent got `Infected`. Both implementations read like a formal specification, again thanks to the declarative nature of functional programming and pattern matching:

```haskell
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
```

The `checkMakeContactInvariants` is a bit more complex but reads as a formal specification as well:

```haskell
checkMakeContactInvariants :: AgentId              -- ^ Agent id of the agent
                           -> [QueueItem SIREvent] -- ^ Events the agent scheduled
                           -> Int                  -- ^ Contact Rate
                           -> Bool
checkMakeContactInvariants sender es contactRate
    -- make sure there has to be exactly one MakeContact event and
    -- exactly contactRate Contact events
    = invOK && hasMakeCont && numCont == contactRate
  where
    (invOK, hasMakeCont, numCont)
      = foldr checkMakeContactInvariantsAux (True, False, 0) es

    checkMakeContactInvariantsAux :: QueueItem SIREvent
                                  -> (Bool, Bool, Int)
                                  -> (Bool, Bool, Int)
    checkMakeContactInvariantsAux
        (QueueItem (Contact sender' Susceptible) receiver t') (b, mkb, n)
      = (b && sender == sender'   -- sender in Contact must be self
           && receiver `elem` ais -- receiver of Contact must be in agent ids
           && t == t', mkb, n+1)  -- Contact event is scheduled immediately
    checkMakeContactInvariantsAux
        (QueueItem MakeContact receiver t') (b, mkb, n)
      = (b && receiver == sender  -- receiver of MakeContact is agent itself
           && t' == t + 1         -- MakeContact scheduled 1 timeunit into future
           &&  not mkb, True, n)  -- there can only be one MakeContact event
```

```haskell
    checkMakeContactInvariantsAux _ (_, _, _)
      = (False, False, 0)         -- other patterns are invalid
```

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.

```haskell
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 $ property $ susceptibleProps evt ao es ai
```

When running this property test all 100 test cases pass. Due to the large random sampling space with 5 parameters, we increase the number of test cases to generate to 100,000 - still all test cases pass.

4.1. Encoding transition probabilities

In the specifications above 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.

4.1.1. Susceptible agent

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.

```haskell
...
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:
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.

4.1.2. Infected agent

We want to write a property test which checks whether the transition from `Infected` to `Recovered` actually follows the exponential distribution with a fixed δ (illness duration). The idea is to compute the expected probability for agents having an illness duration of less or equal δ. This probability is given by the Cumulative Density Function (CDF) of the exponential distribution. The question is how to get the infected illness duration. This is simply achieved by infecting a susceptible agent and taking the scheduling time of the `Recover` event. We have written a custom data generator for this:

```haskell
getInfectedAgentDuration :: Double -> Gen (SIRState, Double)
getInfectedAgentDuration ild = do
  -- with these parameters the susceptible agent WILL become infected
  (_, ao, es) <- genRunSusceptibleAgent 1 1 ild 0 [0] (Contact 0 Infected)
  return (ao, recoveryTime es)
  where
    -- expect exactly one event: Recover
    recoveryTime :: [QueueItem SIREvent] -> Double
    recoveryTime [QueueItem Recover _ t]  = t
    recoveryTime _ = 0
```

Encoding the probability check into a property test is straightforward:

```haskell
prop_infected_duration :: Property
prop_infected_duration = checkCoverage (do
  -- fixed model parameter, otherwise random coverage
  let ild  = 15
  -- compute probability drawing a random value less or equal
  -- ild from the exponential distribution (follows the CDF)
  let prob = 100 * expCDF (1 / ild) ild

  -- run random susceptible agent to become infected and
  -- return agents state and recovery time
  (ao, dur) <- getInfectedAgentDuration ild

  return (cover prob (dur <= ild)
```

```
            ("Infected agent recovery time is less or equals " ++ show ild ++
             ", expected at least " ++ show prob)
            (ao == Infected)) -- final state has to Infected
```

When running the property test we get the following output.

```
+++ OK, passed 3200 tests
    (63.62% Infected agent recovery time is less or equals 15.0,
     expected at least 63.21%).
```

QuickCheck is able to determine after only 3,200 test cases that the expected coverage is met and passes the property test.

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

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 (15) which instead of randomly sampling the test space, enumerates test cases exhaustively up to some depth

References

[1] Asta, S., Özcan, E., and Siebers, P.-O. An investigation on test driven discrete event simulation. In *Operational Research Society Simulation Workshop 2014 (SW14)* (Apr. 2014).

[2] Balci, O. Verification, Validation, and Testing. In *Handbook of Simulation*, J. Banks, Ed. John Wiley & Sons, Inc., 1998, pp. 335–393.

[3] Beck, K. *Test Driven Development: By Example*, 01 edition ed. Addison-Wesley Professional, Boston, Nov. 2002.

[4] Claessen, K., and Hughes, J. QuickCheck - A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 268–279.

[5] Claessen, K., and Hughes, J. Testing Monadic Code with QuickCheck. *SIGPLAN Not. 37*, 12 (Dec. 2002), 47–59.

[6] Collier, N., and Ozik, J. Test-driven agent-based simulation development. In *2013 Winter Simulations Conference (WSC)* (Dec. 2013), pp. 1551–1559.

[7] Epstein, J. M. Chapter 34 Remarks on the Foundations of Agent-Based Generative Social Science. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1585–1604.

[8] Galán, J. M., Izquierdo, L. R., Izquierdo, S. S., Santos, J. I., del Olmo, R., López-Paredes, A., and Edmonds, B. Errors and Artefacts in Agent-Based Modelling. *Journal of Artificial Societies and Social Simulation 12*, 1 (2009), 1.

[9] Hughes, J. QuickCheck Testing for Fun and Profit. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages* (Berlin, Heidelberg, 2007), PADL'07, Springer-Verlag, pp. 1–32.

[10] North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M., and Sydelko, P. Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling 1*, 1 (Mar. 2013), 3.

[11] Onggo, B. S. S., and Karatas, M. Test-driven simulation modelling: A case study using agent-based maritime search-operation simulation. *European Journal of Operational Research 254* (2016), 517–531.

[12] Ormerod, P., and Rosewell, B. Validation and Verification of Agent-Based Models in the Social Sciences. In *Epistemological Aspects of Computer Simulation in the Social Sciences*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Oct. 2006, pp. 130–140.

[13] Polhill, J. G., Izquierdo, L. R., and Gotts, N. M. The Ghost in the Model (and Other Effects of Floating Point Arithmetic). *Journal of Artificial Societies and Social Simulation 8*, 1 (2005), 1.

[14] Robinson, S. *Simulation: The Practice of Model Development and Use.* Macmillan Education UK, Sept. 2014. Google-Books-ID: Dtn0oAEACAAJ.

[15] Runciman, C., Naylor, M., and Lindblad, F. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (New York, NY, USA, 2008), Haskell '08, ACM, pp. 37–48.

[16] Wald, A. Sequential Tests of Statistical Hypotheses. In *Breakthroughs in Statistics: Foundations and Basic Theory*, S. Kotz and N. L. Johnson, Eds., Springer Series in Statistics. Springer New York, New York, NY, 1992, pp. 256–298.