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: 16 Aug 2019 / Accepted: date

Abstract The importance of Agent-Based Simulation (ABS) as scientific method to generate data for scientific models in general and for informed policy decisions in particular has been widely recognised. Due to the implications of the outcomes of an ABS, validation and verification has found its way into this field as well, which is reflected by substantial research on validation and verification, discussing various methods and approaches. However, the important technique of code testing of implementations like unit testing has not generated much research interested so far. We hypothesise that this is due to a conceptual mismatch between the deterministic nature of unit testing and the rather stochastic nature of ABS. As a possible solution, in previous work we have explored the conceptual use of *property-based testing*. In this code testing method, model specifications and invariants are expressed directly in code and tested through *automated* and *randomised* test data generation. This maps much more natural to the stochastic nature of ABS and also scales better than deterministic unit tests which need to be constructed manually for each edge case.

This paper expands on our previous work and explores how to use property-based testing on a technical level to encode and test specifications of ABS. As use case the simple agent-based SIR model is used, where it is shown how to test agent behaviour including transition probabilities. 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 on modern hardware. 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

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

a fundamental part of the model development, supporting hypothesis and discovery making processes. By incorporating this powerful technique into simulation development, 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

1 Introduction

Since its inception in the early 1990s [15,29,34], Agent-Based Simulation (ABS) as a third way of doing science [3,5] has matured substantially and has found its way into the mainstream of science [23]. Further, a number of ABS frameworks and tools like RePast, AnyLogic and NetLogo as well as open databases of ABS models [14] have been developed, allowing for quick and robust prototyping and development of models.

However, despite the broad acceptance and adoption of ABS as methodology and *generative* way of doing science, there have been struggles as described in [4], where the author reports the vulnerability of ABS to misunderstanding. Due to informal specifications of models and change requests amongst members of a research team, bugs are very likely to be introduced. He also reported how difficult it was to reproduce the work of [2], which took the team four months, due to inconsistencies between the original code and the published paper. The consequence is that counter-intuitive simulation results can lead to weeks of checking whether the code matches the model and is bug-free as reported in [3].

The same problem was reported in [20], which tried to reproduce the work of Gintis [17]. In his work, Gintis claimed to have found a mechanism in bilateral decentralized exchange, which resulted in Walrasian General Equilibrium without the neo-classical approach of a tatonement process through a central auctioneer [12]. This was a major breakthrough for economics as the theory of Walrasian General Equilibrium is non-constructive. It postulates the properties and existence of the equilibrium but does not explain the process and dynamics through which this equilibrium can be reached or constructed. Gintis seemed to have found this very process.

The authors [20] failed to reproduce the results and were only able to solve the problem by directly contacting Gintis, which provided the code, the definitive formal reference. It was found that there was a bug in the code leading to unexpected results, which were seriously damaged through this error. They also reported ambiguity between the informal model description in Gintis' paper and the actual implementation. This discovery lead to research in a functional framework for agent-based models of exchange as described in [7], which tried to give a very formal functional specification of the model, coming very close to an implementation in Haskell. The failure of Gintis was investigated in more depth in the thesis by [16] who got access to Gintis' code as well. They found that the code did not follow good object-oriented design principles (all of it was public, code duplication) and - in accordance with [20] - discovered a number of bugs serious enough to invalidate the results.

These issues show that due to the fact that ABS is primarily used for scientific research, producing often break-through scientific results, besides on converging both on standards for testing the robustness of implementations and on its tools, ABS more importantly needs to be *free of bugs*, *verified against their specification*, *validated against hypotheses* and ultimately be *reproducible* [4]. Further, a special issue with ABS is that the emergent behaviour of the system is generally not known in advance and researchers look for some *unique* emergent pattern in the dynamics. Whether the emergent pattern is then truly due to the system working correctly, or a bug in disguise is often not obvious and becomes increasingly difficult to assess with increasing system complexity.

These facts are also underlined by the paper [26] which summarises various ABS development methods, putting fundamental emphasis on the verification and validation (V&V) processes for ABS. Although there exist methods and research of V&V in ABS, unfortunately, as section 2 shows, there does not exist much research on the issue of code testing an ABS implementation. In Software Engineering, this task has been traditionally achieved by unit testing, as introduced by K. Beck in the seminal work on Test Driven Development (TDD) [6]. Unit tests are code pieces which test a given unit of functionality of some given feature. Generally, this results in hundreds or sometimes thousands of unit tests as all execution paths of the software should be covered.

We hypothesise that the reason why unit testing is not very present in the field of ABS V&V research is a conceptual mismatch between unit testings' deterministic and ABS rather stochastic nature. The fact that a unit test needs to be written for each edge case makes it difficult to scale up to the stochastic nature of ABS, where the agent and model behaviour in general is often characterised by probabilistic distributions instead of deterministic rules. As a possible solution to this issue, our work [32] was the first to propose *property-based testing* as an alternative to unit testing for code testing ABS implementations. The main idea of property-based testing is to express model specifications and invariants directly in code and test them through *automated* and *randomised* test data generation. In our paper [32] we presented a few different angles and ways to conceptually use property-based testing to code test ABS implementations. However, we did not discuss technical details and left the exact workings of property-based testing for ABS open as it was beyond the focus of that paper.

In this paper we pick up our conceptual work [32] and put it into a much more technical perspective and demonstrate additional techniques of property-based testing in the context of ABS, which was not covered in the conceptual paper. More specifically, in this paper we show how to encode agent specifications and model invariants into property tests, using an agent-based SIR model [22] as use case. Following an event-driven approach we demonstrate how to express an agent specification in code by relating random input events to specific output events. Further, using specific property-based testing features which allow expressing expected coverage of data distributions, it is shown how transition probabilities can be tested. Finally, we also express model invariants by encoding them into property tests. By doing this, we demonstrate how property-based testing works on a technical level, how specifications and invariants can be put into code and how probabilities can be expressed and tested using statistically robust verification. This underlines the conclusion of our original work [32], that property-based testing maps naturally to ABS. Further, this work shows that in the context of ABS, property-based testing is does scale up better 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: section 2 presents related work. In section 3 property-based testing is introduced on a technical level. In section 4 the agent-based SIR model is introduced, together with its informal event-driven specification. Section 5 contains the main contribution of the paper, where it is shown how to encode agent specifications and transition probabilities with property-based testing. Section 6 shows how to encode model invariants with property-based testing. In section 7 the approach is discussed and related to the work of [32] and other use cases. Finally, section 8 concludes and points out further research.

2 Related Work

Research on code testing of ABS is quite new with few publications so far. Our own work [32] is the first paper to introduce property-based testing to ABS. In it we show on a conceptual level that property-based testing allows to do both verification and validation of an implementation. However, in this work we do not go into technical details of actual implementations nor how to use property-based testing on a technical level.

The work of Collier et al. [13] is the first to discusses how to apply TDD to ABS, using unit testing [6] to verify the correctness of the implementation up to a certain level. They show how to implement unit tests within the RePast Framework and make the important point that such a software needs to be designed to be sufficiently modular otherwise testing becomes too cumbersome and involves too many parts. The authors of [1] discuss a similar approach to DES in the AnyLogic software toolkit.

In [27] the authors propose Test Driven Simulation Modeling (TDSM) which combines techniques from TDD to simulation modeling. The authors present a case study for maritime search operations where they employ ABS. They emphasize that simulation modeling is an iterative process, where changes are made to existing parts, making a TDD approach to simulation modeling a good match. They present how to validate their model against analytical solutions from theory using unit tests by running the whole simulation within a unit test and then perform a statistical comparison against a formal specification. This approach is important for our SIR and Sugarscape case studies.

The paper [8] proposes property-driven design of robot swarms. The authors propose a top-down approach by specifying properties a swarm of robots should have from which a prescriptive model is created, which properties are verified using model checking. Then a simulation is implemented following this prescriptive and verified model after then the physical robots are implemented. The authors identify the main difficulty of implementing such a system that the engineer must *"think at the collective-level, but develop at the individual-level.* It is arguably true that this also applies to implementing agent-based models and simulations where the same collective-individual separation exists from which emergent system behaviour of simulations emerges - this is the very foundation of the ABS methodology.

The authors of [18] give an in-depth and detailed overview over verification, validation and testing of agent-based models and simulations and proposes a generic framework for it. The authors present a generic UML class-model for their framework which they then implement in the two ABS frameworks RePast and MASON. Both of them are implemented in Java and the authors provide a detailed description how their generic testing framework architecture works and how it utilizes JUnit to run automated tests. To demonstrate their framework they provide also a case study of an agent-base simulation of synaptic connectivity where they provide an in-depth explanation of their levels of test together with code.

Although the work on TDD is scarce in ABS, there exists quite some research on applying TDD and unit testing to Multi-Agent Systems (MAS). Although MAS is a different discipline than ABS, the latter one has derived many technical concepts from the former one, thus testing concepts applied to MAS might also be applicable to ABS. [25] performed a survey of testing in MAS. It distinguishes between unit tests of parts that make up an agent, agent tests which test the combined functionality of parts that make up an agent, integration tests which test the interaction of agents within an environment and observe emergent behaviour, system tests which test the MAS as a system running at the target environment and acceptance test in which stakeholders verify that the software meets their goal. Although not all ABS simulations need acceptance and system tests, still this classification gives a good direction and can be directly transferred to ABS.

Property-based testing has a close connection to model-checking [24], where properties of a system are proved in a formal way. The important difference is that the checking happens directly on code and not on the abstract, formal model, thus one can say that it combines model-checking and unit testing, embedding it directly in the software-development and TDD process without an intermediary step. We hypothesise that adding it to the already existing testing methods in the field of ABS is of substantial value as it allows to cover a much wider range of test-cases due to automatic data generation. This can be used in two ways: to verify an implementation against a formal specification and to test hypotheses about an implemented simulation. This puts property-based testing on the same level as agent- and system testing, where not technical implementation details of e.g. agents are checked like in unit tests but their individual complete behaviour and the system behaviour as a whole.

The work of [27] explicitly mention the problem of test coverage, which would often require to write a large number of tests manually to cover the parameter ranges sufficiently enough - property-based testing addresses exactly this problem by *automating* the test-data generation. Note that this is closely related to data-generators [18] and load generators and random testing [9] but property-based testing goes one step further by integrating this into a specification language directly into code, emphasizing a declarative approach and pushing the generators behind the scenes, making them transparent and focusing on the specification rather than on the data-generation.

3 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 [10,11], where the authors present the QuickCheck library, which tries to falsify the specifications by *randomly* sampling the test space. 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 [19].

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.

```haskell
-- 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]
```

QuickCheck generates 100 test cases for each property test 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.

3.1 Generators

QuickCheck comes with a lot of data generators for existing types like `String, Int, Double, [] (List)`, 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 as in the examples above.

Here we implement a custom data generator for the `SIRState` for both cases. We start with the run-time option, running in the `Gen` context:

```
genSIRState :: Gen SIRState
genSIRState = elements [Susceptible, Infected, Recovered]
```

This implementation makes use of the `elements :: [a]` $\rightarrow$ `Gen a` function, 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. Generating on average 80% `Susceptible`, 15% `Infected` and 5% `Recovered` can be achieved using this function:

```
genSIRState :: Gen SIRState
genSIRState = frequency [(80, Susceptible), (15, Infected), (5, Recovered)]
```

Implementing an `Arbitrary` instance is straightforward, one only needs to implement the `arbitrary :: Gen a` method:

```
instance Arbitrary SIRState where
  arbitrary = genSIRState
```

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.

```
newtype Probability = P Double

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

3.2 Distributions

As already mentioned, QuickCheck provides functions to measure the coverage of test cases. This can be done using the `label :: Testable prop ⇒ String → prop → Property` function. It takes a `String` as first argument and a testable property and constructs a `Property`. QuickCheck collects all the generated labels, counts their occurrences and reports their distribution. For example, it could be used to get a rough idea of the length of the random lists created in the `reverse_reverse` property shown above:

```
reverse_reverse_label :: [Int] -> Property
reverse_reverse_label xs
  = label ("length of random-list is " ++ show (length xs))
          (reverse (reverse xs) == xs)
```

When running the test, we see the following output:

```
+++ OK, passed 100 tests:
 5% length of random-list is 27
 5% length of random-list is 0
 4% length of random-list is 19
 ...
```

3.3 Coverage

The most powerful functions to work with test-case distributions though are `cover` and `checkCoverage`. The function `cover :: Testable prop ⇒ Double → Bool → String → prop → Property` allows us 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, and the fourth argument is the property which needs to hold for the test case to succeed.

Here we look at an example where we use `cover` to express that we expect 15% of all test cases to have a random list with at least 50 elements.

```
reverse_reverse_cover :: [Int] -> Property
reverse_reverse_cover xs
  = cover 15 (length xs >= 50) "Length of random list at least 50"
             (reverse (reverse xs) == xs)
```

When repeatedly running the test, we see the following output:

```
+++ OK, passed 100 tests (10% length of random list at least 50).
Only 10% Length of random-list at least 50, but expected 15%.
+++ OK, passed 100 tests (21% length of random list at least 50).
```

As can be seen, QuickCheck runs the default 100 test cases and prints a warning if the expected coverage is not reached. This is a useful feature, but it is up to us to decide whether 100 test cases are suitable and whether we can really claim that the given coverage will be reached or not. Fortunately, QuickCheck provides the powerful function `checkCoverage :: Testable prop ⇒ prop → Property`

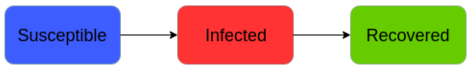

Fig. 1 States and transitions in the SIR compartment model.

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 this, is by using sequential statistical hypothesis testing [33]. 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 us very high confidence in the result.

When we run the example from above but now with `checkCoverage` we get the following output:

```
+++ OK, passed 12800 tests
    (15.445% length of random-list at least 50).
```

We see that after QuickCheck ran 12,800 tests it came to the statistically robust conclusion that, indeed, at least 15% of the test cases have a random list with at least 50 elements.

4 Event-Driven Agent-Based SIR Model

As use case to develop the concepts in this paper, we use the explanatory SIR model [21]. 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 [22] 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 from our repository [30].

4.1 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.

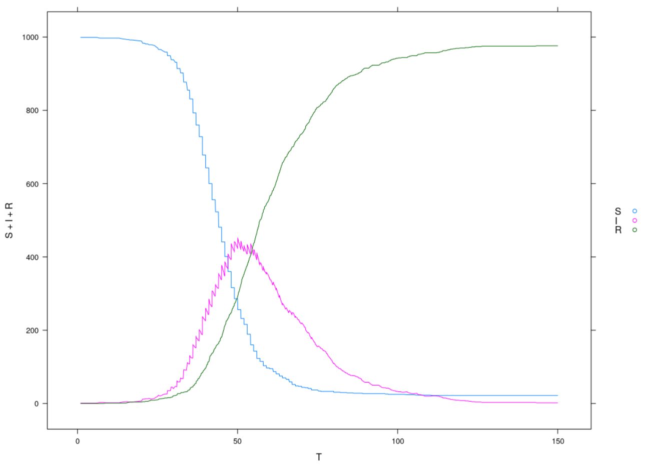

Fig. 2 Dynamics of the SIR compartment model using an event-driven agent-based approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent.

We are using Haskell as notation and implementation as we conducted our research in that language because it originated property-based testing. 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 [31].

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`. Also, 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.

5 Encoding Agent Specifications

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]  -- list of 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 Contact _ Infected AND got infected, check events
susceptibleProps (Contact _ Infected) Infected es ai
  = checkInfectedInvariants ai es
-- received MakeContact => stay Susceptible, check events
susceptibleProps MakeContact Susceptible es ai
  = checkMakeContactInvariants ai es cor
-- all other cases are invalid and result in a failed test case
susceptibleProps _ _ _ _ = False
```

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

```
checkInfectedInvariants :: AgentId               -- agent id of the agent
                        -> [QueueItem SIREvent]  -- list of scheduled events
                        -> 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
```

5.1 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`. We refer to section 3, where we explained the concept of data generators and already implemented all necessary generators for our task at hand.

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 the function `label`, as already explained in section 3, to get an understanding into 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         -- beta (contact rate)
                 -> Probability          -- gamma (infectivity)
                 -> Positive Double      -- delta (illness duration)
                 -> 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 hardware. 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.

5.2 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. To encode these probabilistic properties we are using the function `cover` of QuickCheck. As already introduced in section 3, this function allows us to explicitly specify that a given percentage of successful test cases belong to a given class.

For our case we follow a slightly different approach than in the example of section 3: 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 fixed model parameters can be adjusted arbitrarily and the property must still hold. 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.

```
prop_susceptible_prob :: Positive Double       -- current simulation time
                      -> NonEmptyList AgentId  -- population agent ids
                      -> Property
prop_susceptible_prob (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, only
  -- interested in its output SIRState, ignore id and events
  (_, 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
```

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 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.

As a solution to this fundamental problem, we use QuickChecks `checkCoverage` function. As introduced in section 3, when the function `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. 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 hardware, QuickCheck comes to the statistically robust conclusion that the distributions generated by the test cases reflect the expected distributions and passes the property test.

6 Encoding Model Invariants

By informally reasoning about the agent specification and by realising that they are, in fact, a state machine with a one-directional flow of *Susceptible* → *Infected*

$\rightarrow$ *Recovered*, we can come up with a few invariants which have to hold for any SIR simulation run, *under random model parameters* and independent of the random-number stream and the population:

1. Simulation time is monotonic increasing. Each event carries a timestamp when it is scheduled. This timestamp may stay constant between multiple events but will eventually increase and must never decrease. Obviously, this invariant is a fundamental assumption in most simulations where time advances into the future and does not flow backwards.
2. The number of total agents N stays constant. In the SIR model no dynamic creation or removal of agents during simulation happens. This is in contrast to the Sugarscape where, depending on the model parameters, this can be very well the case.
3. The number of susceptible agents S is monotonic decreasing. Susceptible agents *might* become infected, reducing the total number of susceptible agents but they can never increase because neither an infected nor recovered agent can go back to susceptible.
4. The number of recovered agents R is monotonic increasing. This is because infected agents *will* recover, leading to an increase of recovered agents but once the recovered state is reached, there is no escape from it.
5. The number of infected agents I respects the invariant of the equation $I = N - (S + R)$ for every step. This follows directly from the first property which says $N = S + I + R$.

6.1 Encoding the Invariants

All of those properties are easily expressed directly in code and read like a formal specification due to the declarative nature of functional programming:

```
sirInvariants :: Int                      -- N total number of agents
              -> [(Time,(Int,Int,Int))] -- output each step: (Time,(S,I,R))
              -> Bool
sirInvariants n aos = timeInc && aConst && susDec && recInc && infInv
  where
    (ts, sirs)  = unzip aos
    (ss, _, rs) = unzip3 sirs

    -- 1. time is monotonic increasing
    timeInc = allPairs (<=) ts
    -- 2. number of agents N stays constant in each step
    aConst = all agentCountInv sirs
    -- 3. number of susceptible S is monotonic decreasing
    susDec = allPairs (>=) ss
    -- 4. number of recovered R is monotonic increasing
    recInc = allPairs (<=) rs
    -- 5. number of infected I = N - (S + R)
    infInv = all infectedInv sirs

    agentCountInv :: (Int,Int,Int) -> Bool
    agentCountInv (s,i,r) = s + i + r == n

    infectedInv :: (Int,Int,Int) -> Bool
    infectedInv (s,i,r) = i == n - (s + r)
    allPairs :: (Ord a, Num a) => (a -> a -> Bool) -> [a] -> Bool
    allPairs f xs = all (uncurry f) (pairs xs)

    pairs :: [a] -> [(a,a)]
    pairs xs = zip xs (tail xs)
```

Putting this property into a QuickCheck test is straightforward. We randomise the model parameters β (contact rate), γ (infectivity) and δ (illness duration) because the properties have to hold for all positive, finite model parameters.

```
prop_sir_invariants :: Positive Int    -- beta, contact rate
                    -> Probability     -- gamma, infectivity in range (0,1)
                    -> Positive Double -- delta, illness duration
                    -> TimeRange       -- random duration in range (0, 50)
                    -> [SIRState]      -- population
                    -> Property
prop_sir_invariants
    (Positive beta) (P gamma) (Positive delta) (T t) as  = property (do
  -- total agent count
  let n = length as
  -- run the SIR simulation with a new RNG
  ret <- genSimulationSIR as beta gamma delta t
  -- check invariants and return result
  return (sirInvariants n ret)
```

Due to the large sampling space, we increase the number of test cases to run to 100,000 and all tests pass as expected. It is important to note that we put a random time limit within the range of (0,50) on the simulations to run. Meaning, that if a simulation does not terminate before that limit, it will be terminated at that random `t`. The reason for this is entirely practical as it ensures that the wall clock time to run the tests stays within reasonable bounds while still retaining randomness. In fact, limiting the duration is actually not necessary because we can reason that the SIR simulation *will always* reach an equilibrium in finite steps. TODO: more argument on this detail

7 Discussion

TODO: this is a weak discussion: better structure, bit more details and insights

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 a similar 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.

TODO: first discuss statistical sequential hypothesis testing, before compare it to my conceptual paper

Statistical sequential hypothesis testing can also be applied to the example of hypothesis testing in exploratory models in the paper of [32]. 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*.

8 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. Most importantly, 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 challenge of property-based testing is to write custom data generators, which produce a sufficient coverage for the problem at hand, something not always obvious when starting out. Further, it is not always clear without some analysis whether a property test actually covers enough of the random test space or not. TODO: mention that the use of cover and checkCoverage can help a lot with this issue

As a remedy for the potential coverage problems of QuickCheck, there exists also a *deterministic* property-testing library called SmallCheck of [28], which instead of randomly sampling the test space, enumerates test cases exhaustively up to some depth.

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, 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.

TODO: a bit more?

References

1. Asta, S., Özcan, E., Siebers, P.O.: An investigation on test driven discrete event simulation. In: Operational Research Society Simulation Workshop 2014 (SW14) (2014). URL http://eprints.nottingham.ac.uk/28211/
2. Axelrod, R.: The Convergence and Stability of Cultures: Local Convergence and Global Polarization. Working Paper, Santa Fe Institute (1995). URL http://econpapers.repec.org/paper/wopsafiwp/95-03-028.htm
3. Axelrod, R.: Advancing the Art of Simulation in the Social Sciences. In: Simulating Social Phenomena, pp. 21–40. Springer, Berlin, Heidelberg (1997). DOI 10.1007/978-3-662-03366-1_2. URL https://link.springer.com/chapter/10.1007/978-3-662-03366-1_2
4. Axelrod, R.: Chapter 33 Agent-based Modeling as a Bridge Between Disciplines. In: L.T.a.K.L. Judd (ed.) Handbook of Computational Economics, vol. 2, pp. 1565–1584. Elsevier (2006). DOI 10.1016/S1574-0021(05)02033-2. URL http://www.sciencedirect.com/science/article/pii/S1574002105020332

5. Axelrod, R., Tesfatsion, L.: A Guide for Newcomers to Agent-Based Modeling in the Social Sciences. Staff General Research Papers Archive, Iowa State University, Department of Economics (2006). URL http://econpapers.repec.org/paper/isugenres/12515.htm
6. Beck, K.: Test Driven Development: By Example, 01 edition edn. Addison-Wesley Professional, Boston (2002)
7. Botta, N., Mandel, A., Ionescu, C., Hofmann, M., Lincke, D., Schupp, S., Jaeger, C.: A functional framework for agent-based models of exchange. Applied Mathematics and Computation **218**(8), 4025–4040 (2011). DOI 10.1016/j.amc.2011.08.051. URL http://www.sciencedirect.com/science/article/pii/S0096300311010915
8. Brambilla, M., Pinciroli, C., Birattari, M., Dorigo, M.: Property-driven Design for Swarm Robotics. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1, AAMAS '12, pp. 139–146. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2012). URL http://dl.acm.org/citation.cfm?id=2343576.2343596
9. Burnstein, I.: Practical Software Testing: A Process-Oriented Approach, 1st edn. Springer Publishing Company, Incorporated (2010)
10. 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
11. 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
12. Colell, A.M.: Microeconomic Theory. Oxford University Press (1995). Google-Books-ID: dFS2AQAACAAJ
13. 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
14. ComSES: Computational Model Library (2019). URL https://www.comses.net/codebases/
15. Epstein, J.M., Axtell, R.: Growing Artificial Societies: Social Science from the Bottom Up. The Brookings Institution, Washington, DC, USA (1996)
16. Evensen, P., Märdin, M.: An Extensible and Scalable Agent-Based Simulation of Barter Economics. Master's thesis, Chalmers University of Technology, Göteborg (2010). URL https://gupea.ub.gu.se/handle/2077/22063
17. Gintis, H.: The Emergence of a Price System from Decentralized Bilateral Exchange. Contributions in Theoretical Economics **6**(1), 1–15 (2006). DOI 10.2202/1534-5971.1302. URL https://www.degruyter.com/view/j/bejte.2006.6.1/bejte.2006.6.1.1302/bejte.2006.6.1.1302.xml
18. Gurcan, O., Dikenelli, O., Bernon, C.: A generic testing framework for agent-based simulation models. Journal of Simulation **7**(3), 183–201 (2013). DOI 10.1057/jos.2012.26. URL https://doi.org/10.1057/jos.2012.26
19. 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
20. Ionescu, C., Jansson, P.: Dependently-Typed Programming in Scientific Computing. In: R. Hinze (ed.) Implementation and Application of Functional Languages, Lecture Notes in Computer Science, pp. 140–156. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-41582-1_9. URL http://link.springer.com/chapter/10.1007/978-3-642-41582-1_9
21. 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
22. 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
23. Macal, C.M.: Everything you need to know about agent-based modelling and simulation. Journal of Simulation **10**(2), 144–156 (2016). DOI 10.1057/jos.2016.7. URL https://link.springer.com/article/10.1057/jos.2016.7
24. McMillan, K.L.: Symbolic model checking: An approach to the state explosion problem. Ph.D. thesis, USA (1992). UMI Order No. GAX92-24209

25. Nguyen, C.D., Perini, A., Bernon, C., Pavón, J., Thangarajah, J.: Testing in Multi-agent Systems. In: Proceedings of the 10th International Conference on Agent-oriented Software Engineering, AOSE'10, pp. 180–190. Springer-Verlag, Berlin, Heidelberg (2011). URL http://dl.acm.org/citation.cfm?id=1965954.1965971
26. North, M.J.: Hammer or tongs: How best to build agent-based models? In: Y. Demazeau, B. An, J. Bajo, A. Fernández-Caballero (eds.) Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection, pp. 3–11. Springer International Publishing, Cham (2018)
27. Onggo, B.S.S., Karatas, M.: Test-driven simulation modelling: A case study using agent-based maritime search-operation simulation. European Journal of Operational Research **254**, 517–531 (2016). DOI 10.1016/j.ejor.2016.03.050
28. 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
29. Siebers, P.O., Aickelin, U.: Introduction to Multi-Agent Simulation. arXiv:0803.3905 [cs] (2008). URL http://arxiv.org/abs/0803.3905. ArXiv: 0803.3905
30. Thaler, J.: Repository of Agent-Based SIR implementation in Haskell (2019). URL https://github.com/thalerjonathan/haskell-sir
31. 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
32. Thaler, J., Siebers, P.O.: Show me your properties: The potential of property-based testing in agent-based simulation. In: Proceedings of the 2019 Summer Simulation Conference, SummerSim '19, pp. 1:1–1:12. Society for Computer Simulation International, San Diego, CA, USA (2019). URL http://dl.acm.org/citation.cfm?id=3374138.3374139
33. 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
34. Wooldridge, M.: An Introduction to MultiAgent Systems, 2nd edn. Wiley Publishing (2009)