SHOW ME YOUR PROPERTIES!
THE POTENTIAL OF PROPERTY-BASED TESTING IN AGENT-BASED SIMULATION

Jonathan Thaler
Peer Olaf Siebers

School Of Computer Science
University of Nottingham
7301 Wollaton Rd
Nottingham, United Kingdom
{jonathan.thaler,peer-olaf.siebers}@nottingham.ac.uk

ABSTRACT

This paper presents property-based testing, an approach for testing implementations of agent-based simulations (ABS), never considered so far in this field. It is a complementary technique to unit-testing and allows to test specifications and laws of an implementation directly in code which is then checked using *automated* test-data generation. As case-studies, we present two different models, an agent-based SIR model and the SugarScape model, in which we will show how to apply property-based testing to explanatory and exploratory agent-based models and what its limits are.

Keywords: Agent-Based Simulation, Validation & Verification, Property-Based Testing, Haskell.

1 INTRODUCTION

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

One can argue that ABS should require more rigorous programming standards than other computer simulations (Polhill, Izquierdo, and Gotts 2005). Because researchers in ABS look for an emergent behaviour in the dynamics of the simulation, they are always tempted to look for some surprising behaviour and expect something unexpected from their simulation. Also, due to ABS mostly exploratory nature, there exists some amount of uncertainty about the dynamics the simulation will produce before running it. The authors (Ormerod and Rosewell 2006) see the current process of building ABS as a discovery process where often models of an ABS lack an analytical solution (in general) which makes verification much harder if there is no such solution. Thus it is often very difficult to judge whether an unexpected outcome can be attributed to the model or has in fact its roots in a subtle programming error (Galán, Izquierdo, Izquierdo, Santos, del Olmo, López-Paredes, and Edmonds 2009).

In general this implies that we can only *raise the confidence* in the correctness of the simulation: it is not possible to prove that a model is valid, instead one should think of confidence in its validity. Therefore, the process of V&V is not the proof that a model is correct but the process of trying to prove that the model is incorrect. The more checks one carries out which show that it is not incorrect, the more confidence we can place on the models validity. To tackle such a problem in software, software engineers have developed the concept of test-driven development (TDD).

Test-Driven Development (TDD) was rediscovered in the late 90s by Kent Beck (Beck 2002) as a way to a more agile approach to software-engineering, where instead of doing each step (requirements, implementation, testing,...) as separated from each other, all of them are combined in shorter cycles. Put shortly, in TDD tests are written for each feature before actually implementing it, then the feature is fully implemented and the tests for it should pass. This cycle is repeated until the implementation of all requirements has finished. Traditionally TDD relies on so called unit-tests which can be understood as a piece of code which when run isolated, tests some functionality of an implementation.

Thus we can say that test-driven development in general and unit-testing together with code-coverage in particular, allow to guarantee the correctness of an implementation to some informal degree, which has been proven to be sufficiently enough through years of practice in the software industry all over the world.

In this paper we discuss property-based testing, a complementary method of testing the implementation of an ABS, which allows to directly express model-specifications and laws in code and test them through *automated* test-data generation. We see it as an addition to TDD where it works in combination with unit-testing to verify and validate a simulation to increase the confidence in its correctness.

Property-based testing has its origins (Claessen and Hughes 2000, Claessen and Hughes 2002, Runciman, Naylor, and Lindblad 2008) in the pure functional programming language Haskell (Hudak, Hughes, Peyton Jones, and Wadler 2007) where it was first conceived and implemented. It has been successfully used for testing Haskell code for years and also been proven to be useful in the industry (Hughes 2007). To make this paper sufficiently self-contained we avoid discussing it from a Haskell perspective and present it more on a conceptual level.

We claim that property-based testing is a natural fit to ABS and a valuable addition to the already existing testing methods in this field. To substantiate and test our claims, we present two case-studies. First, the agent-based SIR model (Macal 2010), which is of explanatory nature, where we show how to express formal model-specifications in property-tests. Second, the SugarScape model (Epstein and Axtell 1996), which is of exploratory nature, where we show how to express hypotheses in property-tests and how to property-test agent functionality.

The aim and contribution of this paper is the introduction of property-based testing to ABS. To our best knowledge property-based testing has never been looked at in the context of ABS and this paper is the first one to do so.

The structure of the paper is as follows: First we present related work in Section 2. Then we give a more in-depth explanation of property-based testing in Section 3. Next we shortly discuss how to conceptually apply property-based testing to ABS in Section 4. The heart of the paper are the two case-studies, which we present in Section 5 and 6. Finally we conclude and discuss further research in Section 7.

2 RELATED WORKD

Research on TDD of ABS is quite new and thus there exist relative few publications. The work (Collier and Ozik 2013) is the first to discusses how to apply TDD to ABS, using unit-testing to verify the correctness of the implementation up to a certain level. They show how to implement unit-tests within the RePast Framework (North, Collier, Ozik, Tatara, Macal, Bragen, and Sydelko 2013) 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 paper (Asta, Özcan, and Peer-Olaf 2014) discusses a similar approach to DES in the AnyLogic software toolkit.

The paper (Onggo and Karatas 2016) proposes Test Driven Simulation Modelling (TDSM) which combines techniques from TDD to simulation modelling. The authors present a case study for maritime search-operations where they employ ABS. They emphasise that simulation modelling is an iterative process, where changes are made to existing parts, making a TDD approach to simulation modelling 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 will become of importance later on in our SIR and Sugarscape case studies.

The paper (Gurcan, Dikenelli, and Bernon 2013) gives 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 utilises 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. The paper (Nguyen, Perini, Bernon, Pavón, and Thangarajah 2011) is 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 (McMillan 1993), 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 (Onggo and Karatas 2016) explicitly mentions 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 (Gurcan, Dikenelli, and Bernon 2013) and load generators and random testing (Burnstein 2010) but property-based testing goes one step further by integrating this into a specification language directly into code, emphasising 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 possible. When a case is found for which the property fails, the library then reduces it to the most simple one. 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, which is almost always stochastic by nature, feasible as it does not require the programmer to specify all test-cases by hand, as is required in e.g. traditional unit-tests.

Property-based testing was invented by the authors of (Claessen and Hughes 2000, Claessen and Hughes 2002) in which they present the QuickCheck library in Haskell, which tries to falsify the specifications by *randomly* sampling the space. We argue, that the stochastic sampling nature of this approach is particularly well suited to ABS, because it is itself almost always driven by stochastic events and randomness in the agents behaviour, thus this correlation should make it straight-forward to map ABS to property-testing. The main challenge when using QuickCheck, as will be shown later, is to write *custom* test-data generators for agents and the environment which cover the space sufficiently enough to not miss out on important test-cases. According to the authors of QuickCheck *"The major limitation is that there is no measurement of test coverage."* (Claessen and Hughes 2000). QuickCheck provides help to report the distribution of test-cases but still it could be the case that simple test-cases which would fail are never tested.

To give a rough idea on how property-based testing works in Haskell, 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. Note that the first line of each function defines its name, its inputs (*[Int]* is a list of integers) and the output which is the last type (*Bool*). Note that the *(++)* operator concatenates two lists, *reverse* simply reverses a list.

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

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

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

As a remedy for the potential sampling difficulties of QuickCheck, there exists also a deterministic property-testing library called SmallCheck (Runciman, Naylor, and Lindblad 2008) which instead of randomly sampling the test-space, enumerates test-cases exhaustively up to some depth. It is based on two observations, derived from model-checking, that (1) *"If a program fails to meet its specification in some cases, it almost always fails in some simple case"* and (2) *"If a program does not fail in any simple case, it hardly ever fails in any case* (Runciman, Naylor, and Lindblad 2008). This non-stochastic approach to property-based testing might be a complementary addition in some cases where the tests are of non-stochastic nature with a search-space which is too large to implement manually by unit-tests but is relatively easy and small enough to enumerate exhaustively. The main difficulty and weakness of using SmallCheck is to reduce the dimensionality of the test-case depth search to prevent combinatorial explosion, which would lead to exponential number of cases. Thus one can see QuickCheck and SmallCheck as complementary instead of in opposition to each other. Note that in this paper we only use QuickCheck due to the match of ABS stochastic nature and the random test generation. We refer to SmallCheck in cases where appropriate. Also note that we regard property-based testing as *complementary* to unit-tests and not in opposition - we see it as an addition in the TDD process of developing an ABS.

4 TESTING ABS IMPLEMENTATIONS

Generally we need to distinguish between two types of testing / verification in ABS.

1. Testing / verification of models for which we have real-world data or an analytical solution which can act as a ground-truth - examples for such models are the SIR model, stock-market simulations, social simulations of all kind.
2. Testing / verification of models which are of exploratory nature, inspired by real-world phenomena but for which no ground-truth per se exists - examples for such models is the Sugarscape (Epstein and Axtell 1996) or Agent_Zero model (Epstein 2014).

The baseline is that either one has an analytical model as the foundation of an agent-based model or one does not. In the former case, e.g. the SIR model, one can very easily validate the dynamics generated by the simulation to the one generated by the analytical solution through System Dynamics. In the latter case one has basically no idea or description of the emergent behaviour of the system prior to its execution e.g. SugarScape. In this case it is important to have some hypothesis about the emergent property / dynamics. The question is how verification / validation works in this setting as there is no formal description of the expected behaviour: we don't have a ground-truth against which we can compare our simulation dynamics.

One distinguishes between black-box and white-box verification where in white-box verification one looks directly at code and reasons about it whereas in black-box verification one generally feeds input to the software / functions / methods and compares it to expected output. Black-box verification is our primary concern in this paper as property-based testing is an instance of black-box verification. In the case of ABS we have the following levels of black-box tests:

1. Isolated agent behaviour parts - test the individual parts which make up the agent behaviour under given inputs. For this we can use traditional unit-tests as shown by (Collier and Ozik 2013) and also property-based testing as we will show in the use-cases.
2. Interacting agent behaviour - test if interaction between agents are correct. For this we can use traditional unit-tests as shown by (Collier and Ozik 2013).
3. Simulation dynamics - compare emergent dynamics of the ABS as a whole under given inputs to an analytical solution or real-world dynamics in case there exists some, using statistical tests. We see this type of tests conceptually as property-tests as well because we are testing properties of the model / simulation as we will see in the use-cases. Technically speaking we can both use traditional unit-tests and also property-based tests to implement them - conceptually they are property-tests.
4. Hypotheses - test whether hypotheses about the model are valid or invalid. This is very similar to the previous point but without comparing it to analytical solutions or real-world dynamics but only to some hypothetical values.

5 CASE STUDY I: SIR

As first use-case we discuss property-based testing for the *explanatory* agent-based SIR model. It is a very well studied and understood compartment model from epidemiology (Kermack and McKendrick 1927) which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population. We implemented an agent-based version of this model [1], inspired by (Macal 2010).

In this model, people in a population of size N can be in either one of 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 re-infection, thus these interactions are ignored in this model. Due to the models' origin in System Dynamics (SD) (Porter 1962), there exists a top-down formalisation in SD with the following equations:

$$
\begin{aligned}
\frac{\mathrm{d}S}{\mathrm{d}t} &= -infectionRate \\
\frac{\mathrm{d}I}{\mathrm{d}t} &= infectionRate - recoveryRate \\
\frac{\mathrm{d}R}{\mathrm{d}t} &= recoveryRate
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
infectionRate &= \frac{I \beta S \gamma}{N} \\
recoveryRate &= \frac{I}{\delta}
\end{aligned} \tag{2}
$$

5.1 Deriving a property

Our goal is to derive a property which connects the agent-based implementation to the SD equations. The foundation are both the infection- and recovery-rate where the infection-rate determines how many *Susceptible* agents per time-unit become *Infected* and the recovery-rate determines how many *Infected* agents per time-unit become *Recovered*. Lets look at the algorithm of the susceptible agent behaviour, which is key for the infection-rate:

```
generate on average β make-contact events per time-unit;
if make-contact event then
    select random agent randA from population;
    if agent randA infected then
        become infected with probability γ;
    end
end
```

Algorithm 1: Susceptible behaviour

Per time-unit, a susceptible agent makes *on average* contact with β other agents, where in the case of a contact with an infected agent, the susceptible agent becomes infected with a given probability γ. In this

[1] The code is freely accessible from https://github.com/thalerjonathan/phd/tree/master/public/propabs/sir

description there is another probability hidden, the probability of making contact with an infected agent, which is simply the ratio of number of infected agents to number non-infected agents. We can now derive the formula for the probability of a *Susceptible* agent to become infected: $\beta * \gamma * \frac{number of infected}{number of non-infected}$. When we look at the formula we can see that it is conceptually the same representation of the *infection-rate* of the SD specification as shown above - except that it only considers a single *Susceptible* agent instead of the aggregate of *S* susceptible agents. We have now a property we can check using a property-test.

5.2 Constructing the property-based test

Having a property (law), we want now to construct a property-test for it. The formula is invariant under random population mixes and thus should hold for varying agent populations where the mix of *Susceptible, Infected and Recovered* agents is random - thus we use QuickCheck to generate the population randomly, the property must still hold.

Obviously we need to pay attention to the fact that we are dealing with a stochastic system thus we can only talk about averages and thus it does not suffice to only run a single agent but we are repeating this for e.g. 10.000 *Susceptible* agents (all with different random-number seeds).

To check whether this test has passed we compare the required amount of agents which on average should become infected using the above formula to the one from our tests (simply count the agents which got infected and divide by N) and if the value lies within some small ε then we accept the test as passed. Now we can construct the following property-based test as shown in Algorithm 5.2.

```
input : List randAs of random agent-population generated by QuickCheck
populationCount = length randAs;
infectedCount = count Infected in randAs;
infectionRate = infectivity * contactRate * (infectedCount / populationCount);
susceptibles = create 10000 Susceptible agents;
countInfected = 0;
for each agent sa in susceptibles do
    run agent sa for 1.0 time-unit, with list randAs as input;
    if agent sa became Infected then
        countInfected = countInfected + 1;
    end
end
averageInfectionRate = countInfected / (length susceptibles);
ε = 0.1;
if abs (averageInfectionRate - infectionRate) ≤ ε then
    PASS;
else
    FAIL;
end
```

Algorithm 2: Property-based test for infection-rate.

When running, QuickCheck generates 100 test-cases by randomly generating 100 different *randAs* inputs to the test. All have to pass for the whole property-test to pass, which should be the case with an $\varepsilon = 0.1$.

This is the very power which property-based testing is offering us: we directly express the specification of the original SD model in a test of our agent-based implementation and let QuickCheck generate random test cases for us. This closely ties our implementation to the original specification and raises the confidence to a very high level that it is actually a valid and correct implementation.

6 CASE STUDY II: SUGARSCAPE

We now look at how property-based testing can be made of use in the *exploratory* Sugarscape model (Epstein and Axtell 1996). It was one of the first models in ABS, with the aim to *grow* an artificial society by simulation and connect observations in their simulation to phenomenon observed in real-world societies. In this model a population of agents move around in a discrete 2D environment, where sugar grows, and interact with each other and the environment in many different ways. The main features of this model are (amongst others): searching, harvesting and consuming of resources, wealth and age distributions, population dynamics under sexual reproduction, cultural processes and transmission, combat and assimilation, bilateral decentralized trading (bartering) between agents with endogenous demand and supply, disease processes transmission and immunology. For our research we undertook a *full and validated* implementation of the Sugarscape model [2]. We validated of our implementation against the book (Epstein and Axtell 1996) and a NetLogo implementation (Weaver 2009) [3] during which we also implemented property tests. Due to lack of space we added a discussion of the validation process as an Appendix A.

Whereas in the explanatory SIR case-study we had an analytical solution, inspired by the SD origins of the model, the fundamental difference in the exploratory Sugarscape model is that none such analytical solutions exist. This raises the question, which properties we can actually test in such a model - we propose the following:

- Environment behaviour - the Sugarscape environment has its own behaviour which boils down to regrowing of resources. The correct working can be tested using property-tests by generating random environments and checking laws governing the regrowth.
- Agent behaviour - obviously full agent behaviour could be tested with property-tests, using randomly generated agents (with random values in their properties). As it turned out to be quite difficult to derive properties for, in this paper we restricted ourselves to test parts of agent behaviour and also left out testing of agent interactions.
- Emergent behaviour - although we don't have analytical descriptions of properties of our model in the case of Sugarscape, there still exist informal descriptions and more formal hypotheses about emergent properties. Property-testing can be used to check them and when proved to be valid can be seen as regression tests.

6.1 Environment behaviour

The environment in the Sugarscape model has some very simple behaviour: each site has a sugar level and when harvested by an agent, it regrows back to the full level over time. Depending on the configuration of the model it either grows back immediately within 1 tick or over multiple ticks. We can construct simple property-tests for these behaviours. In the case the sugar grows back immediately we let QuickCheck generate a random environment and then run the environment behaviour for 1 tick and then check the property that all sites have to be back to their maximum sugar level. In the case of regrow over multiple ticks, we also use QuickCheck to generate a random environment but additionally a random *positive* rate (which is a floating point number) which we then use to calculate the number of ticks until full regrowth.

[2] The code can be accessed freely from https://github.com/thalerjonathan/phd/tree/master/public/towards/SugarScape/sequential

[3] https://www2.le.ac.uk/departments/interdisciplinary-science/research/replicating-sugarscape

After running the random environment for the given number of ticks all sites have to be back to full sugar level - see 6.1 for an algorithm for this case.

Note that QuickCheck initially doesn't know how to generate a random environment because each site consists of a custom data-structure for which QuickCheck is not able to generate random instances by default. This problem is solved by writing a custom data-generator, for which existing QuickCheck functions can be used e.g. picking the current sugar level of a site from a random range.

input : Random environment *env* generated by QuickCheck
input : Regrowth rate *randRate* (positive floating point) generated by QuickCheck
maxTicks = maxSugarCapacityOnSites / randRate;
env' = runEnvironmentTicks maxTicks env;
sites = getEnvironmentSites env';
if *all sites maxSugarLevel* **then**
| PASS;
else
| FAIL;
end

Algorithm 3: Property-based test for rate-based regrow of sugar on all sites.

The Sugarscape environment is a torus where the coordinates wrap around in both dimensions. To check whether the implementation of the wrapping-calculation is correct we used both unit- and property-tests. With the unit-tests we carefully constructed all possible cases we could think of and came up with 13 test-cases. With the property-based test we simply defined a single test-case where we expressed the property that after wrapping *any* coordinates, supplied by QuickCheck, the wrapped coordinates have to be within bounds. See the algorithm in 6.1.

input : Random 2d discrete coordinate *randCoord* generated by QuickCheck
(x, y) = wrapCoordinates randCoord;
if $(x \geq 0$ *and* $x \leq environmentDimX)$ *and* $(y \geq 0$ *and* $y \leq environmentDimY)$ **then**
| PASS;
else
| FAIL;
end

Algorithm 4: Property-based test for wrap-coordinates functionality.

6.2 Agent behaviour

We implemented a number of property-tests for agent functions which just cover a part of an agents behaviour: checks whether an agent has died of age or starved to death, the metabolism, immunisation step, check if an agent is a potential borrower or fertile, lookout, trading transactions. We provided custom data-generators for the agents and let QuickCheck generate the random data and us running the agent with the provided data, checking for the properties.

As an example, provided in listing 6.2, we give the property-test of an agent dying from age, which happens when the agents age is greater or equal its maximum age. It might look trivial but property-based testing helps us here to clearly state the invariants (properties) and relieves us from constructing all possible edge-cases because we rely on QuickChecks abilities to cover them for us.

input : Random agent *ag* generated by QuickCheck
input : Random age *randAge* within a specified range generated by QuickCheck
set age of agent ag to randAge;
ageLimit = get agents age limit;
died = agentDieOfAge ag;
if *died == (randAge >= ageLimit* **then**
  PASS;
else
  FAIL;
end

Algorithm 5: Property-based test for agent dying of age.

6.3 Emergent properties

In the validation and verification process of our Sugarscape implementation we put informal descriptions and hypotheses about emergent properties from the Sugarscape book into formal property-tests. Examples for such hypotheses / informal descriptions of emergent properties are e.g. the carrying capacity becomes stable after 100 steps; when agents trade with each other, after 1000 steps the standard deviation of trading prices is less than 0.05; when there are cultures, after 2700 steps either one culture dominates the other or both are equally present.

The property we test for is whether *the emergent property under test is stable under varying random-number seeds* or not. Put another way, we let QuickCheck generate random number streams and require that the tests all pass with arbitrary random number streams. Unfortunately this revealed that this property doesn't hold for all emergent properties. The problem is that QuickCheck generates by default 100 test-cases for each property-test where all need to pass for the whole property-test to pass - this wasn't the case, where most of the 100 test-cases passed but unfortunately not all. Thus in this case a different approach is required: instead of requiring *every* test to pass we require that *most* tests pass, which can be achieved using a t-test with a confidence interval of e.g. 95%. This means we won't use QuickCheck anymore and resort to a normal unit-test where we run the simulation 100 times with different random number streams each time and then performing a t-test with a 95% confidence interval. Note that we are now technically speaking of a unit-test but conceptually it is still a property-test.

In listing 6.3 we show the algorithm of a property-test for checking whether after 1000 steps the standard deviation of trading prices is less than 0.05. The test passes if out of 100 runs a 95% confidence interval is reached using a t-test.

7 CONCLUSIONS

We found property-based testing particularly well suited for ABS firstly due to ABS stochastic nature and second 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 often exploratory nature of ABS.

Although propert-based testing has its origins in Haskell, there exist now libraries for it in other languages e.g. Java, Pyhton, C++ as well and we hope that our research sparked an interest in applying property-based testing to the established object-oriented languages in ABS as well.

We didn't look into testing full agent and interacting agent behaviour using property-tests due to its complexity which would justify a whole paper alone. Due to its inherent stateful nature with complex dependencies

```
maxTicks = 1000;
replications = 100;
stdAverage = 0.05;
tradingPriceStdsList = empty list;
for i ← 1 to replications do
    rng = new random number generator;
    simContext = initSimulation rng;
    out = runSimulation maxTicks simContext;
    tps = extractTradingPrices out;
    tpsStd = calculate standard deviation of tps;
    insert tpsStd into tradingPriceStdsList;
end
tTestPass = perform 1-sided t-test comparing stdAverage with tradingPriceStdsList on a 0.95 interval;
if tTestPass then
    PASS;
else
    FAIL;
end
```

Algorithm 6: Property-based test for trading prices.

between valid states and agents actions we need a more sophisticated approach as outlined in (De Vries 2019), where the authors show how to build a meta-model and commands which allow to specify properties and valid state-transitions which can be generated automatically. We leave this for further research.

ACKNOWLEDGMENTS

The authors would like to thank J. Hey and M. Handley for valuable feedback and discussions.

REFERENCES

Asta, S., E. Özcan, and S. Peer-Olaf. 2014, April. "An investigation on test driven discrete event simulation". In *Operational Research Society Simulation Workshop 2014 (SW14)*.

Balci, O. 1998. "Verification, Validation, and Testing". In *Handbook of Simulation*, edited by J. Banks, pp. 335–393. John Wiley & Sons, Inc.

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

Burnstein, I. 2010. *Practical Software Testing: A Process-Oriented Approach*. 1st ed. Springer Publishing Company, Incorporated.

Claessen, K., and J. Hughes. 2000. "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. New York, NY, USA, ACM.

Claessen, K., and J. Hughes. 2002, December. "Testing Monadic Code with QuickCheck". *SIGPLAN Not.* vol. 37 (12), pp. 47–59.

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

De Vries, Edsko 2019, January. "Well-Typed".

Epstein, J. M. 2014, February. *Agent_Zero: Toward Neurocognitive Foundations for Generative Social Science*. Princeton University Press. Google-Books-ID: VJEpAgAAQBAJ.

Epstein, J. M., and R. Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up*. Washington, DC, USA, The Brookings Institution.

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

Gurcan, O., O. Dikenelli, and C. Bernon. 2013, August. "A generic testing framework for agent-based simulation models". *Journal of Simulation* vol. 7 (3), pp. 183–201.

Hudak, P., J. Hughes, S. Peyton Jones, and P. Wadler. 2007. "A History of Haskell: Being Lazy with Class". In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pp. 12–1–12–55. New York, NY, USA, ACM.

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

Kermack, W. O., and A. G. McKendrick. 1927, August. "A Contribution to the Mathematical Theory of Epidemics". *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* vol. 115 (772), pp. 700–721.

Macal, C. M. 2010. "To Agent-based Simulation from System Dynamics". In *Proceedings of the Winter Simulation Conference*, WSC '10, pp. 371–382. Baltimore, Maryland, Winter Simulation Conference.

McMillan, K. L. 1993. *Symbolic Model Checking*. Norwell, MA, USA, Kluwer Academic Publishers.

Nguyen, C. D., A. Perini, C. Bernon, J. Pavón, and J. Thangarajah. 2011. "Testing in Multi-agent Systems". In *Proceedings of the 10th International Conference on Agent-oriented Software Engineering*, AOSE'10, pp. 180–190. Berlin, Heidelberg, Springer-Verlag.

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

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

Ormerod, P., and B. Rosewell. 2006, October. "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, pp. 130–140. Springer, Berlin, Heidelberg.

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

Porter, D. E. 1962, February. "Industrial Dynamics. Jay Forrester. M.I.T. Press, Cambridge, Mass.; Wiley, New York, 1961. xv + 464 pp. Illus. $18". *Science* vol. 135 (3502), pp. 426–427.

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

Runciman, C., M. Naylor, and F. Lindblad. 2008. "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. New York, NY, USA, ACM.

Weaver, Iain 2009, October. "Replicating Sugarscape in NetLogo".

A VALIDATING SUGARSCAPE IN HASKELL

Obviously we wanted our implementation to be correct, which means we validated it against the informal reports in the book. Also we use the work of (Weaver 2009) [4] which replicated Sugarscape in NetLogo and reported on it [5].

In addition to the informal descriptions of the dynamics, we implemented tests which conceptually check the model for emergent properties with hypotheses shown and expressed in the book. Technically speaking we have implemented that with unit-tests where in general we run the whole simulation with a fixed scenario and test the output for statistical properties which, in some cases is straight forward e.g. in case of Trading the authors of the Sugarscape model explicitly state that the standard deviation is below 0.05 after 1000 ticks. Obviously one needs to run multiple replications of the same simulation, each with a different random-number generator and perform a statistical test depending on what one is checking: in case of an expected mean one utilises a t-test and in case of standard-deviations a chi-squared test.

A.1 Terracing

Our implementation reproduces the terracing phenomenon as described on page TODO in Animation and as can be seen in the NetLogo implementation as well. We implemented a property-test in which we measure the closeness of agents to the ridge: counting the number of same-level sugars cells around them and if there is at least one lower then they are at the edge. If a certain percentage is at the edge then we accept terracing. The question is just how much, which we estimated from tests and resulted in 45%. Also, in the terracing animation the agents actually never move which is because sugar immediately grows back thus there is no incentive for an agent to actually move after it has moved to the nearest largest cite in can see. Therefore we test that the coordinates of the agents after 50 steps are the same for the remaining steps.

A.2 Carrying Capacity

Our simulation reached a steady state (variance < 4 after 100 steps) with a mean around 182. Epstein reported a carrying capacity of 224 (page 30) and the NetLogo implementations' (Weaver 2009) carrying capacity fluctuates around 205 which both are significantly higher than ours. Something was definitely wrong - the carrying capacity has to be around 200 (we trust in this case the NetLogo implementation and deem 224 an outlier).

After inspection of the NetLogo model we realised that we implicitly assumed that the metabolism range is *continuously* uniformly randomized between 1 and 4 but this seemed not what the original authors intended: in the NetLogo model there were a few agents surviving on sugarlevel 1 which was never the case in ours as the probability of drawing a metabolism of exactly 1 is practically zero when drawing from a continuous range. We thus changed our implementation to draw a discrete value as the metabolism.

This partly solved the problem, the carrying capacity was now around 204 which is much better than 182 but still a far cry from 210 or even 224. After adjusting the order in which agents apply the Sugarscape rules, by looking at the code of the NetLogo implementation, we arrived at a comparable carrying capacity of the NetLogo implementation: agents first make their move and harvest sugar and only after this the agents metabolism is applied (and ageing in subsequent experiments).

[4] https://www2.le.ac.uk/departments/interdisciplinary-science/research/replicating-sugarscape

[5] Note that lending didn't properly work in their NetLogo code and that they didn't implement Combat

For regression-tests we implemented a property-test which tests that the carrying capacity of 100 simulation runs lies within a 95% confidence interval of a 210 mean. These values are quite reasonable to assume, when looking at the NetLogo implementation - again we deem the reported Carrying Capacity of 224 in the Book to be an outlier / part of other details we don't know.

One lesson learned is that even such seemingly minor things like continuous vs. discrete or order of actions an agent makes, can have substantial impact on the dynamics of a simulation.

A.3 Wealth Distribution

By visual comparison we validated that the wealth distribution (page 32-37) becomes strongly skewed with a histogram showing a fat tail, power-law distribution where very few agents are very rich and most of the agents are quite poor. We compute the skewness and kurtosis of the distribution which is around a skewness of 1.5, clearly indicating a right skewed distribution and a kurtosis which is around 2.0 which clearly indicates the 1st histogram of Animation II-3 on page 34. Also we compute the Gini coefficient and it varies between 0.47 and 0.5 - this is accordance with Animation II-4 on page 38 which shows a gini-coefficient which stabilises around 0.5 after. We implemented a regression-test testing skewness, kurtosis and gini-coefficients of 100 runs to be within a 95% confidence interval of a two-sided t-test using an expected skewness of 1.5, kurtosis of 2.0 and gini-coefficient of 0.48.

A.4 Migration

With the information provided by (Weaver 2009) we could replicate the waves as visible in the NetLogo implementation as well. Also we propose that a vision of 10 is not enough yet and shall be increased to 15 which makes the waves very prominent and keeps them up for much longer - agent waves are travelling back and forth between both Sugarscape peaks. We haven't implemented a regression-test for this property as we couldn't come up with a reasonable straight forward approach to implement it.

A.5 Pollution and Diffusion

With the information provided by (Weaver 2009) we could replicate the pollution behaviour as visible in the NetLogo implementation as well. We haven't implemented a regression-test for this property as we couldn't come up with a reasonable straight forward approach to implement it.

A.6 Mating

We could not replicate Figure III-1 (TODO: page) - our dynamics first raised and then plunged to about 100 agents and go then on to recover and fluctuate around 300. This findings are in accordance with (Weaver 2009), where they report similar findings - also when running their NetLogo code we find the dynamics to be qualitatively the same.

Also at first we weren't able to reproduce the cycles of population sizes. Then we realised that our agent-behaviour was not correct: agents which died from age or metabolism could still engage in mating before actually dying - fixing this to the behaviour, that agents which died from age or metabolism won't engage in mating solved that and produces the same swings as in (Weaver 2009). Although our bug might be obvious, the lack of specification of the order of the application of the rules is an issue in the SugarScape book.

A.7 Inheritance

We couldn't replicate the findings of the Sugarscape book regarding the Gini coefficient with inheritance. The authors report that they reach a gini coefficient of 0.7 and above in Animation III-4. Our Gini coefficient fluctuated around 0.35. Compared to the same configuration but without inheritance (Animation III-1) which reached a Gini coefficient of about 0.21, this is indeed a substantial increase - also with inheritance we reach a larger number of agents of around 1,000 as compared to around 300 without inheritance. The Sugarscape book compares this to chapter II, Animation II-4 for which they report a Gini coefficient of around 0.5 which we could reproduce as well. The question remains, why it is lower (lower inequality) with inheritance?

The baseline is that this shows that inheritance indeed has an influence on the inequality in a population. Thus we deemed that our results are qualitatively the same as the make the same point. Still there must be some mechanisms going on behind the scenes which are unspecified in the original Sugarscape.

A.8 Cultural Dynamics

We could replicate the cultural dynamics of AnimationIII-6 / Figure III-8: after 2700 steps either one culture (red / blue) dominates both hills or each hill is dominated by a different ulture. We wrote a test for it in which we run the simulation for 2.700 steps and then check if either culture dominates with a ratio of 95% or if they are equal dominant with 45%. Because always a few agents stay stationary on sugarlevel 1 (they have a metabolism of 1 and cant see far enough to move towards the hills, thus stay always on same spot because no improvement and grow back to 1 after 1 step), there are a few agents which never participate in the cultural process and thus no complete convergence can happen. This is accordance with (Weaver 2009).

A.9 Combat

Unfortunately (Weaver 2009) didn't implement combat, so we couldn't compare it to their dynamics. Also, we weren't able to replicate the dynamics found in the Sugarscape book: the two tribes always formed a clear battlefront where some agents engage in combat e.g. when one single agent strays too far from its tribe and comes into vision of the other tribe it will be killed almost always immediately. This is because crossing the sugar valley is costly: this agent wont harvest as much as the agents staying on their hill thus will be less wealthy and thus easier killed off. Also retaliation is not possible without any of its own tribe anywhere near.

We didn't see a single run where an agent of an opposite tribe "invaded" the other tribes hill and ran havoc killing off the entire tribe. We don't see how this can happen: the two tribes start in opposite corners and quickly occupy the respective sugar hills. So both tribes are acting on average the same and also because of the number of agents no single agent can gather extreme amounts of wealth - the wealth should rise in both tribes equally on average. Thus it is very unlikely that a super-wealthy agent emerges, which makes the transition to the other side and starts killing off agents at large. First: a super-wealthy agent is unlikely to emerge, second making the transition to the other side is costly and also low probability, third the other tribe is quite wealthy as well having harvested for the same time the sugar hill, thus it might be that the agent might kill a few but the closer it gets to the center of the tribe the less like is a kill due to retaliation avoidance - the agent will simply get killed by others.

Also it is unclear in case of AnimationIII-11 if the R rule also applies to agents which get killed in combat. Nothing in the book makes this clear and we left it untouched so that agents who only die from age (original R rule) are replaced. This will lead to a near-extinction of the whole population quite quickly as agents kill each other off until 1 single agent is left which will never get killed in combat because there are no other agents who could kill it - instead it will enter an infinite die and reborn cycle thanks to the R rule.

A.10 Spice

The book specifies for AnimationIV-1 a vision between 1-10 and a metabolism between 1-5. The last one seems to be quite strange because the maximum sugar / spice an agent can find is 4 which means that agents with metabolism of either 5 will die no matter what they do because the can never harvest enough to satisfy their metabolism. When running our implementation with this configuration the number of agents quickly drops from 400 to 105 and continues to slowly degrade below 90 after around 1000 steps. The implementation of (Weaver 2009) used a slightly different configuration for AnimationIV-1, where they set vision to 1-6 and metabolism to 1-4. Their dynamics stabilise to 97 agents after around 500+ steps. When we use the same configuration as theirs, we produce the same dynamics. Also it is worth nothing that our visual output is strikingly similar to both the book AnimationIV-1 and (Weaver 2009).

A.11 Trading

For trading we had a look at the NetLogo implementation of (Weaver 2009): there an agent engages in trading with its neighbours *over multiple rounds* until either MRSs cross over or no trade has happened anymore. Because (Weaver 2009) were able to exactly replicate the dynamics of the trading time-series we assume that their implementation is correct. We think that the fact that an agent interact with its neighbours over multiple rounds is made not very clear in the book. The only hint is found on page 102: *"This process is repeated until no further gains from trades are possible."* which is not very clear and does not specify exactly what is going on: does the agent engage with all neighbours again? is the ordering random? Another hint is found on page 105 where trading is to be stopped after MRS cross-over to prevent an infinite loop. Unfortunately this is missing in the Agent trade rule T on page 105. Additional information on this is found in footnote 23 on page 107. Further on page 107: *"If exchange of the commodities will not cause the agents' MRSs to cross over then the transaction occurs, the agents recompute their MRSs, and bargaining begins anew."*. This is probably the clearest hint that trading could occur over multiple rounds.

We still managed to exactly replicate the trading-dynamics as shown in the book in Figure IV-3, Figure IV-4 and Figure IV-5. The book is also pretty specific on the dynamics of the trading-prices standard-deviation: on page 109 the authors specify that at t=1000 the standard deviation will have always fallen below 0.05 (Figure IV-5), thus we implemented a property-test which tests for exactly that property. Unfortunately we didn't reach the same magnitude of the trading volume where ours is much lower around 50 but it is equally erratic, so we attribute these differences to other missing specifications or different measurements because the price-dynamics match that well already so we can safely assume that our trading implementation is correct.

According to the book, Carrying Capacity (Animation II-2) is increased by Trade (page 111/112). To check this it is important to compare it not against AnimationII-2 but a variation of the configuration for it where spice is enabled, otherwise the results are not comparable because carrying capacity changes substantially when spice is on the environment and trade turned off. We could replicate the findings of the book: the carrying capacity increases slightly when trading is turned on. Also does the average vision decrease and the average metabolism increase. This makes perfect sense: trading allows genetically weaker agents to survive which results in a slightly higher carrying capacity but shows a weaker genetic performance of the population.

According to the book, increasing the agent vision leads to a faster convergence towards the (near) equilibrium price (page 117/118/119, Figure IV-8 and Figure IV-9). We could replicate this behaviour as well.

According to the book, when enabling R rule and giving agents a finite life span between 60 and 100 this will lead to price dispersion: the trading prices won't converge around the equilibrium and the standard deviation will fluctuate wildly (page 120, Figure IV-10 and Figure IV-11). We could replicate this behaviour as well.

The Gini coefficient should be higher when trading is enabled (page 122, Figure IV-13) - We could replicate this behaviour.

Finite lives with sexual reproduction lead to prices which don't converge (page 123, Figure IV-14). We could reproduce this as well but it was important to re-set the parameters to reasonable values: increasing number of agents from 200 to 400, metabolism to 1-4 and vision to 1-6, most important the initial endowments back to 5-25 (both sugar and spice) otherwise hardly any mating would happen because the agents need too much wealth to engage (only fertile when have gathered more than initial endowment). What was kind of interesting is that in this scenario the trading volume of sugar is substantially higher than the spice volume - about 3 times as high.

From this part, we didn't implement: Effect of Culturally Varying Preferences, page 124 - 126, Externalities and Price Disequilibrium: The effect of Pollution, page 126 - 118, On The Evolution of Foresight page 129 / 130.

A.12 Diseases

We were able to exactly replicate the behaviour of Animation V-1 and Animation V-2: in the first case the population rids itself of all diseases (maximum 10) which happens pretty quickly, in less than 100 ticks. In the second case the population fails to do so because of the much larger number of diseases (25) in circulation. We used the same parameters as in the book. The authors of (Weaver 2009) could only replicate the first animation exactly and the second was only deemed "good". Their implementation differs slightly from ours: In their case a disease can be passed to an agent who is immune to it - this is not possible in ours. In their case if an agent has already the disease, the transmitting agent selects a new disease, the other agent has not yet - this is not the case in our implementation and we think this is unreasonable to follow: it would require too much information and is also unrealistic. We wrote regression tests which check for animation V-1 that after 100 ticks there are no more infected agents and for animation V-2 that after 1000 ticks there are still infected agents left and they dominate: there are more infected than recovered agents.