# 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, a novel approach of testing implementations of agent-based simulations (ABS). 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. We conducted our research in the pure functional programming language Haskell, which is the origin of property-based testing. Besides being especially suited for property-based testing, it supports strong isolation in unit-tests and controlled side-effects which further increases isolation of tests and allows extremely convenient checking of invariants.

**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).

Put shortly, in TDD one writes to called tests for each feature before actually implementing it. One should implement as many tests as necessary to fully test the functionality of that feature. Then the feature is implemented and the tests for it should pass. This cycle is repeated until the implementation of all requirements has finished. Of course it is important to cover the whole functionality with tests to be sure that all cases are checked which can be supported by code coverage tools to ensure that all code-paths have been tested. Traditionally TDD relies on so called unit-tests which can be understood as a piece of code which when run, tests some functionality of an implementation. Each unit-test should only test a small, clearly distinguishable part of the implementation and should not depend on other tests e.g. which need to run before or after - it should be able to run isolated.

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 and thus we discuss it from that perspective. It has been successfully used for testing Haskell code for years and also been proven to be useful in the industry (Hughes 2007).

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 **??**. 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 need 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 case study.

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 which tests units that make up an agent, agent tests which test the combined functionality of units that make up an agent, integration tests which test the interaction of agents within an environment and observe emergent behaviour, system test 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 or 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 with some user-defined coverage. 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, 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 as already explained. 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.

```
-- 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

in ABS depending on which level we are, property-based testing means different things and does not necessarily involve QuickCheck and can also be technically implemented as unit-tests

Generally we need to distinguish between two types of testing/verification: 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 and 2. testing/verification of models which are just exploratory and which are only be inspired by real-world phenomena - examples for such models are Epsteins Sugarscape and Agent_Zero.

So 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.

### 4.1 Black-Box Verification

In black-box Verification one generally feeds input and compares it to expected output. In the case of ABS we have the following examples of black-box test:

1. Isolated Agent Behaviour - test isolated agent behaviour under given inputs using and property-based testing.
2. Interacting Agent Behaviour - test if interaction between agents are correct .
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.
4. Hypotheses- test whether hypotheses are valid or invalid using and property-based testing.

Using black-box verification and property-based testing we can apply for the following use cases for testing ABS in FRP:

### 4.2 White-Box Verification

White-Box verification is necessary when we need to reason about properties like *forever*, *never*, which cannot be guaranteed from black-box tests. Additional help can be coverage tests with which we can show that all code paths have been covered in our tests.

TODO: List of Common Bugs and Programming Practices to avoid them (Vipindeep and Jalote 2005)

We have discussed in this section *how* to approach an ABS implementation from a pure functional perspective using Haskell where we have also briefly touched on *why* one should do so and what the benefits and drawbacks are. In the next two sections we will expand on the *why* by presenting two case-studies which show the benefits of using Haskell in regards of testing and increasing the confidence in the correctness of the implementation.

## 5 CASE STUDY I: SIR

As an example we discuss the black-box testing for the SIR model using property-testing. We test if the *isolated* behaviour of an agent in all three states Susceptible, Infected and Recovered, corresponds to model specifications. The crucial thing though is that we are dealing with a stochastic system where the agents act *on averages*, which means we need to average our tests as well. We conducted the tests on the implementation found in the paper of Appendix **??**.

**Finding optimal $\Delta t$** The selection of the right $\Delta t$ can be quite difficult in FRP because we have to make assumptions about the system a priori. One could just play it safe with a very conservative, small $\Delta t < 0.1$ but the smaller $\Delta t$, the lower the performance as it multiplies the number of steps to calculate. Obviously one wants to select the *optimal* $\Delta t$, which in the case of ABS is the largest possible $\Delta t$ for which we still get the correct simulation dynamics. To find out the *optimal* $\Delta t$ one can make direct use of the black-box tests: start with a large $\Delta t = 1.0$ and reduce it by half every time the tests fail until no more tests fail - if for $\Delta t = 1.0$ tests already pass, increasing it may be an option. It is important to note that although isolated agent behaviour tests might result in larger $\Delta t$, in the end when they are run in the aggregate system, one needs to sample the whole system with the smallest $\Delta t$ found amongst all tests. Another option would be to apply super-sampling to just the parts which need a very small $\Delta t$ but this is out of scope of this paper.

**Agents as signals** Agents *might* behave as signals in FRP which means that their behaviour is completely determined by the passing of time: they only change when time changes thus if they are a signal they should stay constant if time stays constant. This means that they should not change in case one is sampling the system with $\Delta t = 0$. Of course to prove whether this will *always* be the case is strictly speaking impossible with a black-box verification but we can gain a good level of confidence with them also because we are staying pure. It is only through white-box verification that we can really guarantee and prove this property.

### 5.0.1 Black-Box Verification

The interface of the agent behaviours are defined below. When running the SF with a given $\Delta t$ one has to feed in the state of all the other agents as input and the agent outputs its state it is after this $\Delta t$.

```
data SIRState
  = Susceptible
  | Infected
  | Recovered

type SIRAgent = SF [SIRState] SIRState

susceptibleAgent :: RandomGen g => g -> SIRAgent
infectedAgent :: RandomGen g => g -> SIRAgent
recoveredAgent :: SIRAgent
```

**Susceptible Behaviour** A susceptible agent *may* become infected, depending on the number of infected agents in relation to non-infected the susceptible agent has contact to. To make this property testable we run a susceptible agent for 1.0 time-unit (note that we are sampling the system with a smaller $\Delta t = 0.1$) and then check if it is infected - that is it returns infected as its current state.

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. $N = 10.000$ agents (all with different RNGs). We then need a formula for the required fraction of the N agents which should have become infected on average. Per 1.0 time-unit, a susceptible agent makes *on average* contact with $\beta$ other agents where in the case of a contact with an infected agent the susceptible agent becomes infected with a given probability $\gamma$. In this description there is another probability hidden, which is the probability of making contact with an infected agent which is simply the ratio of number of infected agents to number not infected agents. The formula for the target fraction of agents which become infected is then: $\beta * \gamma * \frac{numberofinfected}{numberofnon-infected}$. To check whether this test has passed we compare the required amount of agents which on average should become infected 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 $\varepsilon$ then we accept the test as passed.

Obviously the input to the susceptible agents which we can vary is the set of agents with which the susceptible agents make contact with. To save us from constructing all possible edge-cases and combinations and testing them with unit-tests we use property-testing with QuickCheck which creates them randomly for us and reduces them also to all relevant edge-cases. This is an example for how to use property-based testing in ABS where QuickCheck can be of immense help generating random test-data to cover all cases.

**Infected Behaviour** An infected agent *will always* recover after a finite time, which is *on average* after $\delta$ time-units. Note that this property involves stochastics too, so to test this property we run a large number of infected agents e.g. $N = 10.000$ (all with different RNGs) until they recover, record the time of each agents recovery and then average over all recovery times. To check whether this test has passed we compare the average recovery times to $\delta$ and if they lie within some small $\varepsilon$ then we accept the test as passed.

We use property-testing with QuickCheck in this case as well to generate the set of other agents as input for the infected agents. Strictly speaking this would not be necessary as an infected agent never makes contact with other agents and simply ignores them - we could as well just feed in an empty list. We opted for using QuickCheck for the following reasons:

- We wanted to stick to the interface specification of the agent-implementation as close as possible which asks to pass the states of all agents as input.
- We shouldn't make any assumptions about the actual implementation and if it REALLY ignores the other agents, so we strictly stick to the interface which requires us to input the states of all the other agents.
- The set of other agents is ignored when determining whether the test has failed or not which indicates by construction that the behaviour of an infected agent does not depend on other agents.
- We are not just running a single replication over 10.000 agents but 100 of them which should give black-box verification more strength.

**Recovered Behaviour** A recovered agent will stay in the recovered state *forever*. Obviously we cannot write a black-box test that truly verifies that because it had to run in fact forever. In this case we need to resort to white-box verification (see below).

Because we use multiple replications in combination with QuickCheck obviously results in longer test-runs (about 5 minutes on my machine) In our implementation we utilized the FRP paradigm. It seems that functional programming and FRP allow extremely easy testing of individual agent behaviour because FP and FRP compose extremely well which in turn means that there are no global dependencies as e.g. in OOP where we have to be very careful to clean up the system after each test - this is not an issue at all in our *pure* approach to ABS.

**Simulation Dynamics** We won't go into the details of comparing the dynamics of an ABS to an analytical solution, that has been done already by (Macal 2010). What is important is to note that population-size matters: different population-size results in slightly different dynamics in SD => need same population size in ABS (probably...?). Note that it is utterly difficult to compare the dynamics of an ABS to the one of a SD approach as ABS dynamics are stochastic which explore a much wider spectrum of dynamics e.g. it could be the case, that the infected agent recovers without having infected any other agent, which would lead to an extreme mismatch to the SD approach but is absolutely a valid dynamic in the case of an ABS. The question is then rather if and how far those two are *really* comparable as it seems that the ABS is a more powerful system which presents many more paths through the dynamics.

**Finding optimal $\Delta t$** Obviously the *optimal* $\Delta t$ of the SIR model depends heavily on the model parameters: contact rate $\beta$ and illness duration $\delta$. We fixed them in our tests to be $\beta = 5$ and $\delta = 15$. By using the isolated behaviour tests we found an optimal $\Delta t = 0.125$ for the susceptible behaviour and $\Delta t = 0.25$ for the infected behaviour.

**Agents as signals** Our SIR agents *are* signals due to the underlying continuous nature of the analytical SIR model and to some extent we can guarantee this through black-box testing. For this we write tests for each individual behaviour as previously but instead of checking whether agents got infected or have recovered we assume that they stay constant: they will output always the same state when sampling the system with $\Delta t = 0$. The tests are conceptual the complementary tests of the previous behaviour tests so in conjunction with them we can assume to some extent that agents are signals. To prove it, we need to look into white-box verification as we cannot make guarantees about properties which should hold *forever* in a computational setting.

### 5.0.2 White-Box Verification

In the case of the SIR model we have the following invariants:

- A susceptible agent will *never* make the transition to recovered.
- An infected agent will *never* make the transition to susceptible.
- A recovered agent will *forever* stay recovered.

All these invariants can be guaranteed when reasoning about the code. An additional help will be then coverage testing with which we can show that an infected agent never returns susceptible, and a susceptible agent never returned infected given all of their functionality was covered which has to imply that it can never occur!

We will only look at the recovered behaviour as it is the simplest one. We leave the susceptible and infected behaviours for further research / the final thesis because the conceptual idea becomes clear from looking at the recovered agent.

**Recovered Behaviour** The implementation of the recovered behaviour is as follows:

```
recoveredAgent :: SIRAgent
recoveredAgent = arr (const Recovered)
```

Just by looking at the type we can guarantee the following:

- it is pure, no side-effects of any kind can occur
- no stochasticity possible because no RNG is fed in / we don't run in the random monad

The implementation is as concise as it can get and we can reason that it is indeed a correct implementation of the recovered specification: we lift the constant function which returns the Recovered state into an arrow. Per definition and by looking at the implementation, the constant function ignores its input and returns always the same value. This is exactly the behaviour which we need for the recovered agent. Thus we can reason that the recovered agent will return Recovered *forever* which means our implementation is indeed correct.

## 6 CASE STUDY II: SUGARSCAPE

We implemented a number of tests for agent functions which don't cover a whole sub-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 transaction. What all these functions have in common is that they are not pure computations like utility functions but require an agent-continuation which means they have access to the agent state, environment and random-number stream. This allows testing to capture the *complete* system state in one location, which allows the checking of much more invariants than in approaches which have implicit side-effects. What is particularly powerful is that one has complete control and insight over the changed state before and after e.g. a function was called on an agent: thus it is very easy to check if the function just tested has changed the agent-state itself or the environment: the new environment is returned after running the agent and can be checked for equality of the initial one - if the environments are not the same, one simply lets the test fail. This behaviour is very hard to emulate in OOP because one can not exclude side-effect at compile time, which means that some implicit data-change might slip away unnoticed. In FP we get this for free.

We tested these functions with an approach called *property-based* testing. Although it is now available in a wide range of programming languages and paradigms, property-based testing has its origins in Haskell (Claessen and Hughes 2000, Claessen and Hughes 2002) and we argue that for that reason it really shines in pure functional programming. Property-based testing allows to formulate *functional specifications* in code which then the property-testing library (e.g. QuickCheck (Claessen and Hughes 2000)) tries to falsify by automatically generating random test-data covering as much cases as possible. When an input is found for which the property fails, the library then reduces it to the most simple one.

We implement custom data-generators for our agent state and environment and its cells and then let QuickCheck generate the random data and us running the agent with the provided data, checking for the properties. An example for such a property is that an agent has starved to death in case its sugar (or spice) level has dropped to 0. The corresponding property-test generates a random agent state and also a random sugar level which we set in the agent state. We then run the function which returns True in case the agent has starved to death. We can then check that this flag is true only if the initial random sugar level was less then or equal 0.

We found that property-based testing works surprisingly well in this context because properties seem to be quite abound here. Also, it is clear to see that this kind of testing is especially well suited to 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 (again the declarative nature of functional programming shines through). Also the deductive nature of falsification in property-based testing suits very well the constructive nature of ABS.

Further we undertook a full validation of our implementation against the book and TODO CITE WEAVER for which we also implemented property tests but on the simulation / model level:

## 7 CONCLUSIONS

We found property-based testing particularly well suited for ABS. Although it is now available in a wide range of programming languages and paradigms, propert-based testing has its origins in Haskell (Claessen and Hughes 2000, Claessen and Hughes 2002) and we argue that for that reason it really shines in pure functional programming. Property-based testing allows to formulate *functional specifications* in code which then the property-testing library (e.g. QuickCheck (Claessen and Hughes 2000)) tries to falsify by automatically generating random test-data covering as much cases as possible. When an input 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 (again the declarative nature of functional programming shines through). Also the deductive nature of falsification in property-based testing suits very well the constructive nature of ABS.

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

### 7.1 Further Research

TODO

## ACKNOWLEDGMENTS

The authors would like to thank J. Hey 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.

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.

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.

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.

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.

Vipindeep, V., and P. Jalote. 2005, March. "List of Common Bugs and Programming Practices to avoid them". Technical report, Indian Institute of Technology, Kanpur.

Weaver, I. "Replicating Sugarscape in NetLogo". Technical report.

## 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 ) which replicated Sugarscape in NetLogo and reported on it [1].

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 reproduce 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, this 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 need 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 ) 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 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).

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

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

looking at NetLogo - 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 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 ) 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 ) 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

Initially 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 ), 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 ). 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 ).

### A.9 Combat

Unfortunately (Weaver ) 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 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 ) 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 ).

### A.11 Trading

For trading we had a look at the NetLogo implementation of (Weaver ): there an agent engages in trading with its neighbours *over multiple rounds* until MRSs cross over and no trade has happened anymore. Because (Weaver ) were able to exactly replicate the dynamics of the trading time-series we assume that their implementation is correct. Unfortunately 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 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 and the test passed. 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 ) 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.