HANDS OFF MY PROPERTY!

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 a new and complementary approach to unit-testing the implementation of agent-based simulations, called property-based testing which allows to test specifications and laws of the implementation directly in code which is then tested using *automated* test-data generation. We present two different models as case-studies in which we will show how to apply property-based testing to exploratory and explanatory agent-based models and what its limits are.

We conduct our implementations in the pure functional programming language Haskell, which is the origin of property-based testing. Also we show that simply by switching to such a language one gets rid of a large class of run-time bugs and is able to make stronger guarantees of correctness already at compile time without writing tests for some parts. Further, it makes isolated unit-tests quite easier.

Keywords: Agent-Based Simulation, Property-Based Testing, Validation & Verification, Model Checking, 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. 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 trying to prove that the model is incorrect. The more tests/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) (todo: cite).

TODO: dont spend so much on TDD, mention that property-based testing can be put in the TDD framework but is not the focus of this paper Test-Driven Development (TDD) was conceived in the late 90s by Kent Beck (TODO: cite) 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. TDD approaches software construction in a way that one writes first unit-tests for the functionality one wants to test and then iteratively implements this functionality until all tests succeed. This is then repeated until the whole software package is finished. The important difference to traditional models, where the steps are done in separation from each other, is that the customer receives a working software package at the end of each short cycle, allowing to change requirements which in turn allows the software-development team to react quickly to changing requirements.

It is important to understand that the unit-tests act both as documentation / specification of what the code / interface which is tested should do and as an insurance against future changes which might break existing code. If the tests cover all possible code paths - there exist tools to measure the test-coverage and visualising the missing code-paths / tests - of the software, then if the tests also succeed after future changes one has very high confidence that these future changes didn't break existing functionality. If though tests break then either the changes are erroneous or the tests are an incomplete specification and need to be adapted to the new features.

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. Also a fundamental strength of such tests is that programmers gain much more confidence when making changes to code - without such tests all bets are off and there is no reliable way to know whether the changes have broken something or not.

TODO: find out what property-based testing makes different from parameter variation - is there any difference? The work of (Figueredo, Siebers, Owen, Reps, and Aickelin 2014) compares System Dynamics approaches to Cancer Cell simulations to ABS. They investigate the statistical differences between a deterministic ODE (ordinary differential equation), a stochastic Gillespie Algorithm and a stochastic ABS approach. The authors reported that the larger the population size is the closer the 3 approaches matches, especially ABS matches the System Dynamics approaches more closely. Still they found that ABS is capable of capturing "rare" patterns which is due to the memory and stochastic variability on the individual level.

In this paper we discuss a complementary method of testing the implementation of an ABS, called *property-based* testing, which allows to directly express model-specifications 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), thus we investigate its potential for ABS, which to to our best knowledge has not been done yet.

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.

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.

The aim and contribution of this paper is the investigation of the potential of pure functional property-based testing for ABS using Haskell as programming language. Further we will show that by simply using a pure functional programming language removes a large class of run-time errors and allows much stronger guarantees of correctness already at compile time, increasing the confidence in the correctness of the simulation up to a new level.

The structure of the paper is as follows: First we present related work in Section 2. To make this paper sufficiently self-contained, we introduce pure functional programming in Haskell on a conceptual level in Section **??**. Then we give a more in-depth explanation of property-based testing in Section 3. Next we shortly present existing research on *how* to implement ABS in Haskell and conceptually apply property-based testing in Section **??**. The heart of the paper are the two case-studies, which we present in Section 5 and 6. Finally we conclude in Section 7 and discuss further research in Section 8.

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 the TDD approach 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 (Brambilla, Pinciroli, Birattari, and Dorigo 2012) propose property-driven design of robot swarms. They propose a top-down approach by specifying properties a swarm of robots should have from which a prescriptive model is created, which properties are verified using model checking. Then a simulation is implemented following this prescriptive and verified model after then the physical robots are implemented. The authors identify the main difficulty of implementing such a system that the engineer must *"think at the collective-level, but develop at the individual-level*. It is arguably true that this also applies to implementing agent-based models and simulations where the same collective-individual separation exists from which emergent system behaviour of simulations emerges - this is the very foundation of the ABS methodology.

The 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.

The review of the literature in the field gives the impression, that most research focuses on high-level validation and does not deal too much with verification on a technical, code-base level.

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.

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 feasible as it does not require the programmer to specify all test-cases by hand, as is required in 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.

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 primarily focus on the use of 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 VERIFYING ABS IMPLEMENTATIONS

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.

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 Δt The selection of the right Δt 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 Δt, the lower the performance as it multiplies the number of steps to calculate. Obviously one wants to select the *optimal* Δt, which in the case of ABS is the largest possible Δt for which we still get the correct simulation dynamics. To find out the *optimal* Δt 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 Δt, in the end when they are run in the aggregate system, one needs to sample the whole system with the smallest Δt found amongst all tests. Another option would be to apply super-sampling to just the parts which need a very small Δt 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 Δt one has to feed in the state of all the other agents as input and the agent outputs its state it is after this Δt.

```
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 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 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 ε 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 δ 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 δ and if they lie within some small ε 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 Δt Obviously the *optimal* Δt of the SIR model depends heavily on the model parameters: contact rate β and illness duration δ. 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.

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.

8 FURTHER RESEARCH

TODO

ACKNOWLEDGMENTS

The authors would like to thank

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.

Brambilla, M., C. Pinciroli, M. Birattari, and M. Dorigo. 2012. "Property-driven Design for Swarm Robotics". In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '12, pp. 139–146. Richland, SC, International Foundation for Autonomous Agents and Multiagent Systems.

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.

Figueredo, G. P., P.-O. Siebers, M. R. Owen, J. Reps, and U. Aickelin. 2014, April. "Comparing Stochastic Differential Equations and Agent-Based Modelling and Simulation for Early-Stage Cancer". *PLOS ONE* vol. 9 (4), pp. e95150.

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.

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.