Hands Off My Property!

Debugging And Property-Based Testing Of

Pure Functional Agent-Based Simulations

Jonathan Thaler
jonathan.thaler@nottingham.ac.uk
University of Nottingham
Nottingham, United Kingdom

ABSTRACT

This paper presents a new approach to test the implementation of agent-based simulations, called property-based testing which allows to test specifications much more directly than unit tests. Although being more expressive than unit tests, property-based testing is seen as a complementary and does not make unit-testing obsolete. 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-pased testing. 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.

TODO: this would be ideal to submit to an ABS conference so i can also discuss functional programming

TODO: write related research TODO: write background TODO: implement case study 1: property-testing of SIR TODO: implement case study 2: property-testing of Sugarscape TODO: write conclusion & further research

KEYWORDS

Agent-Based Simulation, Property-Based Testing, Model Checking, Haskell

ACM Reference Format:
Jonathan Thaler. 2019. Hands Off My Property!: Debugging And Property-Based Testing Of Pure Functional Agent-Based Simulations. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'18).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When implementing an agent-based simulation (ABS) it is of utmost importance that the implementation is correct up to a specification, which is the model. To ensure that an implementation matches a specification, one uses verification, which ensures that *"we are building the model right"* (TODO: cite). With the established approaches in the field of ABS, which are primarily the object-oriented programming languages Java, Python and C++, it is very difficult, or might even be impossible to *formally* prove that an implementation is correct up to a specification. Also one can say in general that formal proofs of correctness are highly complex and take a lot of effort and are almost always beyond the scope of a project and just not feasible. Still, not checking the correctness of the implementation in *some* way would be highly irresponsible and thus software engineers have developed the concept of unit-testing and test-driven development (todo: cite). Put shortly, in test-driven development one implements unit-tests for each feature to implement before actually implementing the feature. Then the features 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. 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. The work of [7] discusses the use of test-driven development with unit-tests to implement ABS and we support the position that every ABS which has some degree of complexity and is used for some form of policy or decision making should be thoroughly tested.

In this paper we discuss a complementary method of testing the implementation of an ABS, called property-based testing, which to our best knowledge has not been discussed in the field of ABS yet. We present two models for case-studies. First, the SIR model, which is of explanatory nature, where we show how to express formal model-specifications in property-tests. Second, the SugarScape model, which is exploratory nature, where we show how to express hypotheses in property-tests.

Property-based testing has its origins in the pure functional programming language Haskell (TODO: cite quickcheck papers and history of haskell paper) so we discuss it from that perspective. We show that besides property-based testing, Haskell has a lot more to offer in regards to testing and guaranteeing the correctness of an ABS implementation. Simply by switching to this language removes a large class of run-time bugs and allows to make stronger guarantees about the correctness of the ABS implementation, making the implementation *very likely* to be correct.

The aim of this paper is to investigate the potential of property-based testing which allows to directly express model-specifications in code and test them.

The contributions of this paper are:

IFL'18, August 2019, Lowell, MA, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Showing that by simply switching to a pure functional programming language removes a large class of run-time errors and allows much stronger guarantees of correctness already at compile time and without actual need of many tests which are necessary in imperative object-oriented languages. Also we give a brief overview of how to debug time-driven agent-based simulations in such a language using library Haskell-Titan for debugging time-driven agent-based simulations.
- Expanding the testing techniques of ABS implementations by property-based testing, an additional, highly expressive method from which we hope it makes testing easier and ABS implementations which utilise it more likely to be correct.

The structure of the paper is as follows. To make this paper sufficiently self-contained we give a brief introduction of the concepts of pure functional programming in Haskell in Section TODO. Then we introduce property-based testing in general in Section TODO. Then we present both case-studies in Section TODO and Section TODO. We follow with related work in Section TODO. Finally we conclude and give further research in Section TODO.

2 BACKGROUND

2.1 Verification & Validation

The book [25] discusses various aspects of testing and making sure a simulation is correct and distinguishes in this process between *validation* and *verification*. It defines validation to be the process of ensuring that a model or specification is sufficiently accurate for the purpose at hand whereas verification to be the process of ensuring that the model design has been transformed into a computer model with sufficient accuracy. In other words, [3] define validation as *"are we building the right model?"* and verification as *"are we building the model right?"*.

In this paper we will only focus on verification, because it is there where one ensures that the model is programmed correctly, the algorithms have been implemented properly, and the model does not contain errors, oversights, or bugs. Note that verification has a narrow definition and can be seen as a subset of the wider issue of validation. One distinguishes between:

- White-box Verification: compares the content of the model to the *conceptual* model by detailed, micro check if each part of the implementation represent the conceptual model with sufficient accuracy.
- Black-box Verification: treating the functionality to test as a black-box with inputs and outputs and comparing controlled inputs to expected outputs.

So in general one can see verification as a test of the fidelity with which the conceptual model is converted into the computer model. Verification is a continuous process and if it is already there in the programming language / supported by then this is much easier to do. A fundamental hypothesis of this paper is that by choosing a programming language which supports this continuous verification and validation process, then the result is an implementation of a model which is more likely to be correct.

Unfortunately, there is no such thing as general validity: a model should be built for one purpose as simple as possible and not be too general, otherwise it becomes too bloated and too difficult or impossible to analyse. Also, there may be no real world to compare against: simulations are developed for proposed systems, new production or facilities which don't exist yet. Further, it is questionable which real world one is speaking of: the real world can be interpreted in different ways, therefore a model valid to one person might not be valid to another. Sometimes validation struggles because the real world data are inaccurate or there is not enough time to verify and validate everything.

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 verification and validation 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.

In our research we focus primarily on the *Verification* aspect of agent-based simulation: ensuring that the implementation reflects the specifications of the *conceptual* model - have we built the model right? Thus we are not interested in our research into making connections to the real world and always see the model specifications as our "last resort", our ground truth beyond nothing else exists. When there are hypotheses formulated, we always treat and interpret them in respect of the conceptual model.

The authors [21] make the important point that because the current process of building ABS is a discovery process, often models of an ABS lack an analytical solution (in general) which makes verification much harder if there is no such solution.

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.

The paper [24] makes the very important point that ABS should require more rigorous programming standards than other computer simulations. 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. Thus it is of utmost importance to ensure that the implementation is matching the specification as closely as possible and nothing is left to chance. The paper [8] supports this as well and emphasises that the fundamental problem of ABS is that due to its 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.

The work of [13] suggests good programming practice which is extremely important for high quality code and reduces bugs but real world practice and experience show that this alone is

not enough, even the best programmers make mistakes which often can be prevented through a strong static or a dependent type system already at compile-time. What we can guarantee already at compile-time, doesn't need to be checked at run-time which saves substantial amount of time as at run-time there may be a large number of execution paths through the simulation which is almost always simply not feasible to check (note that we also need to check all combinations). This paper also cites modularity as very important for verification: divide and conquer and test all modules separately. We claim that this is especially easy in functional programming as code composes better than in traditional object-oriented programming due to the lack of interdependence between data and code as in objects and the lack of global mutable state (e.g. class variables or global variables) - this makes code extremely convenient to test. The paper also discusses statistical tests (the t test) to check if the outcome of a simulation is sufficiently close to real-world dynamics - we explicitly omit this as it part of validation and not the focus of this research.

2.2 Test-Driven Development

Test-Driven Development (TDD) was conceived in the late 90s by Kent Beck (TODO: cite) as an 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 e.g. the waterfall model 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.

The work of [7] discusses how to apply the test-driven development approach to ABS, using unit-testing to check the correctness of the implementation up to a certain level. The paper [2] discusses a similar approach to DES in the AnyLogic software toolkit which also supports ABS and thus we claim can be applied to ABS as well. We experienced when doing the literature review of this paper that while there exists quite some work on validating an ABS, there doesn't exist much work on verification which discusses the problem from an implementation perspective with program code with these two papers being the only exception.

2.3 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 [5, 6] 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."* [5]. 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 [26] 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* [26]. 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.

2.4 Pure Functional Programming

Although property-based is now available in a wide range of programming languages and paradigms, including Java, Python and

C++, it has its origins in Haskell and indeed both QuickCheck and SmallCheck are Haskell libraries. We argue that for that reason property-based testing really shines in pure functional programming, thus we conduct all implementation and research of this paper using Haskell. Therefore we give a brief introduction into the concepts of pure functional programming in Haskell without going into too much technical detail. Further we will show that the use of Haskell automatically increases the confidence in the correctness of an ABS implementation due to its fundamentally different nature. Also it emphasises loose coupled programming to a much stronger extent that does object-oriented programming as in Java, Python and C++, something the authors of [7] emphasise to be able to properly test agent behaviour. We argue that due to its fundamental different nature, the functional programming paradigm makes making mistakes much harder, resulting in simulations which are more likely to be correct than implementations with existing object-oriented approaches.

Functional programming makes functions the main concept of programming, promoting them to first-class citizens. Its roots lie in the Lambda Calculus which was first described by Alonzo Church [4]. This is a fundamentally different approach to computation than imperative and object-oriented programming which roots lie in the Turing Machine [28]. Rather than describing *how* something is computed as in the more operational approach of the Turing Machine, due to the more declarative nature of the Lambda Calculus, code in functional programming describes *what* is computed.

In this paper we are using the functional programming language Haskell. The paper of [9] gives a comprehensive overview over the history of the language, how it developed and its features and is very interesting to read and get accustomed to the background of the language. The main points why we decided to go for Haskell are:

- Rich Feature-Set - it has all fundamental concepts of the pure functional programming paradigm of which we explain the most important below.
- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications [9], is applicable to a number of real-world problems [22] and has a large number of libraries available [1].
- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science. Further, the community is the main source of high-quality libraries.

As a short example we give an implementation of the factorial function in Haskell:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

When looking at this function we can already see the central concepts of functional programming:

(1) Declarative - we describe *what* the factorial function is rather than how to compute it. This is supported by *pattern matching* which allows to give multiple equations for the same function, matching on its input.
(2) Immutable data - in functional programming we don't have mutable variables - after a variable is assigned, it cannot change its contents. This also means that there is no destructive assignment operator which can re-assign values to a variable. To change values, we employ recursion.
(3) Recursion - the function calls itself with a smaller argument and will eventually reach the case of 0. Recursion is the very meat of functional programming because they are the only way to implement loops in this paradigm due to immutable data.
(4) Static Types - the first line indicates the name and the types of the function. In this case the function takes one Integer as input and returns an Integer as output. Types are static in Haskell which means that there can be no type-errors at run-time e.g. when one tries to cast one type into another because this is not supported by this kind of type-system.
(5) Explicit input and output - all data which are required and produced by the function have to be explicitly passed in and out of it. There exists no global mutable data whatsoever and data-flow is always explicit.
(6) Referential transparency - calling this function with the same argument will *always* lead to the same result, meaning one can replace this function by its value. This means that when implementing this function one can not read from a file or open a connection to a server. This is also known as *purity* and is indicated in Haskell in the types which means that it is also guaranteed by the compiler.

It may seem that one runs into efficiency-problems in Haskell when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of [20] showed that when approaching this problem from a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

For an excellent and widely used introduction to programming in Haskell we refer to [11]. Other, more exhaustive books on learning Haskell are [1, 14]. For an introduction to programming with the Lambda-Calculus we refer to [18]. For more general discussion of functional programming we refer to [9, 10, 16].

2.4.1 Side-Effects. One of the fundamental strengths of functional programming and Haskell is their way of dealing with side-effects in functions. A function with side-effects has observable interactions with some state outside of its explicit scope. This means that the behaviour depends on history and that it loses its referential transparency character, which makes understanding and debugging much harder. Examples for side-effects are (amongst others): modifying a global variable, modifying a variable through a reference, await an input from the keyboard, read or write to a file, open a connection to a server, drawing random-numbers,...

Obviously, to write real-world programs which interact with the outside-world we need side-effects. Haskell allows to indicate in the *type* of a function that it does or does *not* have side-effects. Further there are a broad range of different effect types available, to restrict the possible effects a function can have to only the required type. This is then ensured by the compiler which means

[1] https://wiki.haskell.org/Applications_and_libraries

that a program in which one tries to e.g. read a file in a function which only allows drawing random-numbers will fail to compile. Haskell also provides mechanisms to combine multiple effects e.g. one can define a function which can draw random-numbers and modify some global data. The most common side-effect types are: *IO* allows all kind of I/O related side-effects: reading/writing a file, creating threads, write to the standard output, read from the keyboard, opening network-connections, mutable references,...; *Rand* allows to draw random-numbers from a random-number stream, *Reader* allows to read from a read-only environment, *Writer* allows to write to a write-only environment, *State* allows to read and write a read/write environment.

A function with side-effects has to indicate this in their type e.g. if we want to give our factorial function for debugging purposes the ability to write to the standard output, we add IO to its type: factorial :: Integer -> IO Integer. A function without any side-effect type is called *pure*. A function with a given effect-type needs to be executed with a given effect-runner which takes all necessary parameters depending on the effect and runs a given effectful function returning its return value and depending on the effect also an effect-related result. For example when running a function with a State-effect one needs to specify the initial environment which can be read and written. After running such a function with a State-effect the effect-runner returns the changed environment in addition with the return value of the function itself. Note that we cannot call functions of different effect-types from a function with another effect-type, which would violate the guarantees. Calling a *pure* function though is always allowed because it has by definition no side-effects. An effect-runner itself is a *pure* function. The exception to this is the IO effect type which does not have a runner but originates from the *main* function which is always of type IO.

Although it might seem very restrictive at first, we get a number of benefits from making the type of effects we can use explicit. First we can restrict the side-effects a function can have to a very specific type which is guaranteed at compile time. This means we can have much stronger guarantees about our program and the absence of potential errors already at compile-time which implies that we don't need test them with e.g. unit-tests. Second, because effect-runners are themselves *pure*, we can execute effectful functions in a very controlled way by making the effect-context explicit in the parameters to the effect-runner. This allows a much easier approach to isolated testing because the history of the system is made explicit.

For a technical, in-depth discussion of the concept of side-effects and how they are implemented in Haskell using Monads, we refer to the following papers: [12, 19, 30–32].

2.4.2 Why. In general, Types guide us in program construction by restricting the operations we can perform on the data. This means that by choosing types this reveals already a lot of our program and data and prevents us from making mistakes e.g. interpreting some binary data as text instead of a number. In strongly statically typed languages the types can do this already at compile-time which allows to rule out certain bugs already at compile-time. In general, we can say that for all bugs which can be ruled out at compile-time, we don't need to write property- or unit-tests, because those bugs cannot - per definition - occur at run-time, so it won't make sense to test their absence at run-time. Also, as Dijkstra famously put it: "Testing shows the presence, not the absence of bugs" - thus by induction we can say that compile-time guarantees save us from a potentially infinite amount of testing.

In general it is well established, that pure functional programming as in Haskell, allows to express much stronger guarantees about the correctness of a program *already at compile-time*. This is in fundamental contrast to imperative object-oriented languages like Java or Python where only primitive guarantees about types - mostly relationships between type-hierarchies - can be expressed at compile-time which directly implies that one needs to perform much more testing (user testing or unit-testing) at *run-time* to check whether the model is sufficiently correct. Thus guaranteeing properties already at compile-time frees us from writing unit-tests which cover these cases or test them at run time because they are *guaranteed to be correct under all circumstances, for all inputs.*

In this regards we see pure functional programming as truly superior to the traditional object oriented approaches: they lead to implementations of models which are more likely correct because we can express more guarantees already at compile-time which directly leads to less bugs which directly increases the probability of the software being a correct implementation of the model.

In the next section we give a brief discussion of *how* to apply pure functional programming with Haskell to implement ABS.

3 PURE FUNCTIONAL ABS

There does not exist much research on how to implement an ABS from a pure functional perspective using Haskell. The authors of [27] probably provide the most in-depth and technical discussion by showing *how* to do it and what the benefits and drawbacks are. Their approach is based on Functional Reactive Programming which allows to express discrete- and continuous-time systems in functional programming. It primarily emphasises a time-driven approach to ABS but also allows to implement an event-driven one [17]. Following the conclusions of the paper, we can derive the following benefits, which support directly our initial hypothesis and our claims, giving good reasons *why* to do ABS in a functional way and why it makes it more likely to be correct:

(1) Run-Time robustness by compile-time guarantees - by expressing stronger guarantees already at compile-time we can restrict the classes of bugs which occur at run-time by a substantial amount due to Haskell's strong and static type system. This implies the lack of dynamic types and dynamic casts [2] which removes a substantial source of bugs. Note that we can still have run-time bugs in Haskell when our functions are partial.
(2) Purity - By being explicit and polymorphic in the types about side-effects and the ability to handle side-effects explicitly in a controlled way allows to rule out non-deterministic side-effects which guarantees reproducibility due to guaranteed same initial conditions and deterministic computation. Also by being explicit about side-effects e.g. Random-Numbers and State makes it easier to verify and test.
(3) Explicit Data-Flow and Immutable Data - All data must be explicitly passed to functions thus we can rule out implicit

[2] Note that there exist casts between different numerical types but they are all safe and can never lead to errors at run-time.

data-dependencies because we are excluding IO. This makes reasoning of data-dependencies and data-flow much easier as compared to traditional object-oriented approaches which utilize pointers or references.

(4) Declarative - describing *what* a system is, instead of *how* (imperative) it works. In this way it should be easier to reason about a system and its (expected) behaviour because it is more natural to reason about the behaviour of a system instead of thinking of abstract operational details.

(5) Concurrency and parallelism - due to its pure and 'stateless' nature, functional programming is extremely well suited for massively large-scale applications as it allows adding parallelism without any side-effects and provides very powerful and convenient facilities for concurrent programming. We have explored this more in-depth in Chapter **??**.

Although pure functional ABS as in Haskell allows us to leverage on the concepts of functional and its benefits (and drawbacks) we still rely heavily on (property-based) testing to ensure correctness of a simulation because our approach still can have run-time bugs.

3.1 Debugging

TODO: haskell-titan TODO: Testing and Debugging Functional Reactive Programming [23]

General there are the following basic verification & validation requirements to ABS [25], which all can be addressed in our *pure* functional approach as described in the paper in Appendix **??**:

- Fixing random number streams to allow simulations to be repeated under same conditions - ensured by *pure* functional programming and Random Monads
- Rely only on past - guaranteed with *Arrowized* FRP
- Bugs due to implicitly mutable state - reduced using pure functional programming
- Ruling out external sources of non-determinism / randomness - ensured by *pure* functional programming
- Deterministic time-delta - ensured by *pure* functional programming
- Repeated runs lead to same dynamics - ensured by *pure* functional programming

3.2 Property-Based ABS Testing

TODO: general approach to property-based testing in ABS

Although (pure) functional programming allows us to have stronger guarantees about the behaviour and absence of bugs of the simulation already at compile-time, we still need to test all the properties of our simulation which we cannot guarantee at compile-time.

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 [5, 6] 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 [5]) 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.

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.

3.2.1 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:

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.

3.2.2 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 [29]

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.

4 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 **??**.

4.0.1 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 [15]. 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.

4.0.2 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.

5 CASE STUDY II: SUGARSCAPE

TODP

6 CONCLUSIONS

TODO

7 FURTHER RESEARCH

TODO

Generally speaking, a dependent type is a type whose definition depends on a value e.g. when we have a pair of natural numbers where the second one is greater than the first, we speak of a dependent type. With this power, they allow to push compile-time guarantees to a new level where we can express nearly arbitrary complex guarantees at compile-time because we can *compute types at compile-time*. This means that types are first-class citizen of the language and go as far as being formal proofs of the correctness of an implementation, allowing to narrow the gap between specification and implementation substantially.

We hypothesise that the use of dependent types allows us to push the judgement of the correctness of a simulation to new, unprecedented level, not possible with the established object-oriented approaches so far. This has the direct consequence that the development process is very different and can reduce the amount of testing (both unit-testing and manual testing) substantially. Because one is implementing a simulation which is (as much as possible) correct-by-construction, the correctness (of parts) can be guaranteed statically.

Summarizing, we expect the following benefits from adding dependent types to ABS:

(1) Narrowing the gap between the model specification and its implementation reduces the potential for conceptual errors in model-to-code translation.
(2) Less number of tests required due to guarantees being expressed already at compile time.
(3) Higher confidence in correctness due to formal guarantees in code.

ACKNOWLEDGMENTS

The authors would like to thank

REFERENCES

[1] Christopher Allen and Julie Moronuki. 2016. *Haskell Programming from First Principles.* Allen and Moronuki Publishing. Google-Books-ID: 5FaXDAEACAAJ.

[2] Shahriar Asta, Ender ÃŰzcan, and Siebers Peer-Olaf. 2014. An investigation on test driven discrete event simulation. In *Operational Research Society Simulation Workshop 2014 (SW14)*. http://eprints.nottingham.ac.uk/28211/

[3] Osman Balci. 1998. Verification, Validation, and Testing. In *Handbook of Simulation*, Jerry Banks (Ed.). John Wiley & Sons, Inc., 335–393. https://doi.org/10.1002/9780470172445.ch10

[4] Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 2 (April 1936), 345–363. https://doi.org/10.2307/2371045

[5] Koen Claessen and John 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)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[6] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. *SIGPLAN Not.* 37, 12 (Dec. 2002), 47–59. https://doi.org/10.1145/636517.636527

[7] N. Collier and J. Ozik. 2013. Test-driven agent-based simulation development. In *2013 Winter Simulations Conference (WSC)*. 1551–1559. https://doi.org/10.1109/WSC.2013.6721538

[8] JosÃĺ Manuel GalÃąn, Luis R. Izquierdo, Segismundo S. Izquierdo, JosÃĺ Ignacio Santos, Ricardo del Olmo, Adolfo LÃşpez-Paredes, and Bruce Edmonds. 2009. Errors and Artefacts in Agent-Based Modelling. *Journal of Artificial Societies and Social Simulation* 12, 1 (2009), 1. http://jasss.soc.surrey.ac.uk/12/1/1.html

[9] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip 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)*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[10] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[11] Graham Hutton. 2016. *Programming in Haskell*. Cambridge University Press. Google-Books-ID: 1xHPDAAAQBAJ.

[12] Simon Peyton Jones. 2002. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*. Press, 47–96.

[13] Jack P. C. Kleijnen. 1995. Verification and validation of simulation models. *European Journal of Operational Research* 82, 1 (April 1995), 145–162. https://doi.org/10.1016/0377-2217(94)00016-6

[14] Miran Lipovaca. 2011. *Learn You a Haskell for Great Good!: A Beginner's Guide* (1 edition ed.). No Starch Press, San Francisco, CA.

[15] Charles M. Macal. 2010. To Agent-based Simulation from System Dynamics. In *Proceedings of the Winter Simulation Conference (WSC '10)*. Winter Simulation Conference, Baltimore, Maryland, 371–382. http://dl.acm.org/citation.cfm?id=2433508.2433551

[16] Bruce J. MacLennan. 1990. *Functional Programming: Practice and Theory*. Addison-Wesley. Google-Books-ID: JqhQAAAAMAAJ.

[17] Ruth Meyer. 2014. Event-Driven Multi-agent Simulation. In *Multi-Agent-Based Simulation XV (Lecture Notes in Computer Science)*. Springer, Cham, 3–16. https://doi.org/10.1007/978-3-319-14627-0_1

[18] Greg Michaelson. 2011. *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation. Google-Books-ID: gKvwPtvsSjsC.

[19] E. Moggi. 1989. Computational Lambda-calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Press, Piscataway, NJ, USA, 14–23. http://dl.acm.org/citation.cfm?id=77350.77353

[20] Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA.

[21] Paul Ormerod and Bridget Rosewell. 2006. Validation and Verification of Agent-Based Models in the Social Sciences. In *Epistemological Aspects of Computer Simulation in the Social Sciences*. Springer, Berlin, Heidelberg, 130–140. https://doi.org/10.1007/978-3-642-01109-2_10

[22] Bryan O'Sullivan, John Goerzen, and Don Stewart. 2008. *Real World Haskell* (1st ed.). O'Reilly Media, Inc.

[23] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 2:1–2:27. https://doi.org/10.1145/3110246

[24] J. Gary Polhill, Luis R. Izquierdo, and Nicholas M. Gotts. 2005. The Ghost in the Model (and Other Effects of Floating Point Arithmetic). *Journal of Artificial Societies and Social Simulation* 8, 1 (2005), 1. http://jasss.soc.surrey.ac.uk/8/1/5.html

[25] Stewart Robinson. 2014. *Simulation: The Practice of Model Development and Use*. Macmillan Education UK. Google-Books-ID: Dtn0oAEACAAJ.

[26] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1411286.1411292

[27] Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2019. Pure Functional Epidemics - An Agent-Based Approach. In *International Symposium on Implementation and Application of Functional Languages*. Lowell, Massachusettes.

[28] A. M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

[29] V. Vipindeep and Pankaj Jalote. 2005. *List of Common Bugs and Programming Practices to avoid them*. Technical Report. Indian Institute of Technology, Kanpur.

[30] Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

[31] Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, London, UK, UK, 24–52. http://dl.acm.org/citation.cfm?id=647698.734146

[32] Philip Wadler. 1997. How to Declare an Imperative. *ACM Comput. Surv.* 29, 3 (Sept. 1997), 240–263. https://doi.org/10.1145/262009.262011

Received May 2018