ARTICLE TEMPLATE

Specification Testing of Agent-Based Simulation using Property-Based Testing.

Jonathan Thaler [a] and Peer-Olaf Siebers[a]

[a]School Of Computer Science, University of Nottingham, 7301 Wollaton Rd, Nottingham, UK;

ARTICLE HISTORY
Compiled July 22, 2019

ABSTRACT
This paper explores how to use random property-based testing on a technical level to encode and test specifications of agent-based simulations (ABS). The claim is that opposed to unit testing, random property-based testing is a much more natural fit to test ABS due to both stochastic nature. The paper shows how to test full agent- and model-specifications, in the case of an agents behaviour, its transition probabilities and model invariants. The outcome are specifications expressed directly in code, which relate whole classes of random input to expected classes of output. During test execution, random test data is generated automatically, potentially covering the equivalent of thousands of unit tests, run within seconds. The expressiveness and power of property-based testing is not only limited to be part of a test-driven development process where it acts as specification, verification and regression test but can be integrated as a fundamental part of the model development process, supporting the hypothesis and discovery making process. By incorporating this powerful technique into the simulation development process the confidence in the correctness of an implementation increases dramatically, something of fundamental importance for ABS in general and for ABS supporting far-reaching policy decisions in particular.

KEYWORDS
Testing; Test Driven Development; Model Specification;

1. Introduction

When implementing an 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 (14). In other words, validation determines if we are we building the *right model*, and verification if we are building the *model right* up to some specification (2).

One can argue that ABS should require more rigorous programming standards than other computer simulations (13). Due to the fact that researchers in ABS are looking for an emergent behaviour in the dynamics of the simulation, they are always tempted to look for surprising behaviour and expect something unexpected from their simula-

CONTACT Jonathan Thaler. Email: jonathan.thaler@nottingham.ac.uk

tion. Also, due to ABS' *constructive* and *exploratory* nature (7**?**), there exists some uncertainty about the dynamics the simulation will produce before running it. The authors (12) see the current process of building ABS as a discovery process, where models of an ABS often lack an analytical solution in general, which makes verification much harder if there is no such solution. Thus it is often very difficult to judge whether an unexpected outcome can be attributed to the model or has in fact its roots in a subtle programming error (8).

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

TDD was popularised in the early 00s by Kent Beck (3) as a way to a more agile approach to software engineering, where instead of doing each step (requirements, implementation, testing, delivery) as separated from each other, all of them are combined in shorter cycles. Put shortly, in TDD tests are written for each feature before actually implementing it, then the feature is fully implemented and the tests for it should pass. This cycle is repeated until the implementation of all requirements has finished. Traditionally, TDD relies on so called unit tests which can be understood as a piece of code which when run isolated, tests some functionality of an implementation. Thus we can say that test-driven development in general and unit testing together with some measure of code coverage in particular, guarantee the correctness of an implementation up to some informal degree, which has been proven to be sufficiently enough through years of practice in the software industry all over the world.

TODO cite myself

The work (6) was the first to discuss how to apply TDD to ABS, using unit testing to verify the correctness of the implementation up to a certain level. They show how to implement unit tests within the RePast Framework (10) and make the important point that such a software needs to be designed to be sufficiently modular otherwise testing becomes too cumbersome and involves too many parts. The paper (1) discusses a similar approach to Discrete Event Simulation in the AnyLogic software toolkit.

The paper (11) 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.

Thus, in this chapter we introduce an additional technique for TDD, *property-based testing*, which can be seen as complementary to unit testing. Property-based testing has its origins in Haskell (4; 5; 15), where it was first conceived and implemented. It has been successfully used for testing Haskell code for years and also been proven to be useful in the industry (9). We show and discuss how this technique can be applied to test pure functional ABS implementations. To our best knowledge property-based testing has never been looked at in the context of ABS and this thesis is the first one to do so.

The main idea of property-based testing is to express model specifications and laws directly in code and test them through *automated* and *randomised* test data generation. Thus, a central hypothesis of this thesis is that due to ABS' *stochastic*, *exploratory*, *generative* and *constructive* nature, property-based testing is a natural fit for testing ABS in general and pure functional ABS implementations in particular. It thus should pose a valuable addition to the already existing testing methods in this field, worth exploring.

To substantiate and test our hypothesis, we conducted a few case studies. First, we look into how to express and test agent specifications for both the time- and event-driven SIR implementations in Chapter **??**. Then we show how to encode model invariants of the SIR implementation and validate it against the formal specification from System Dynamics using property tests in Chapter **??**. We explicitly exclude obvious applications of property-based testing, for example boundary checks of the environment and helper functions of agents. Although they are used within an ABS implementation, there is nothing new in testing them, and we rather apply property-based testing to specification and on the model level.

aim of the paper is to build on the rather conceptual paper by me at summersim19 and show property-based testing in ABS on a much more technical level by the case studies of: - encoding agent-specifications of time- and event-driven ABS SIR into property-based tests - random event sampling - compare two different implementations - and encode model invariants

hypothesise that a strong reason for why testing in ABS is not very widely used and adopted is that unit testing is not able to deal very well with the stochastic nature of ABS in general. random property-based testing is a remedy to that problem as it allows to relate whole classes of inputs to specific classes of output for which then randomised test cases are automatically generated, covering potentially thousands of unit tests.

TODO: it would be great if i can show how property-based testing found a bug in an implementation

2. Property-based testing

Property-based testing allows to formulate *functional specifications* in code which then a property-based testing library tries to falsify by *automatically* generating test data, covering as much cases as possible. When a case is found for which the property fails, the library then reduces the test data to its simplest form for which the test still fails, for example shrinking a list to a smaller size. It is clear to see that this kind of testing is especially suited to ABS, because 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 because it does not require the programmer to specify all test cases by hand, as is required in traditional unit tests.

Property-based testing was introduced in (4; 5) where the authors present the QuickCheck library in Haskell, which tries to falsify the specifications by *randomly* sampling the test space. According to the authors of QuickCheck *"The major limitation is that there is no measurement of test coverage."* (4). Although QuickCheck provides help to report the distribution of test cases it is not able to measure the coverage of tests in general. This could lead to the case that test cases which would

fail are never tested because of the stochastic nature of QuickCheck. Fortunately, the library provides mechanisms for the developer to measure coverage in specific test cases where the data and its expected distribution is known to the developer. This is a powerful tool for testing randomness in ABS as will be shown in the next chapters.

As a remedy for the potential coverage problems of QuickCheck, there exists also a deterministic property-testing library called SmallCheck (15), 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* (15). 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 too large to test manually by unit testing but 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.

2.1. A brief overview of QuickCheck

To give a good understanding of how property-based testing works with QuickCheck, we give a few examples of property tests on lists, which are directly expressed as functions in Haskell. Such a function has to return a `Bool` which indicates `True` in case the test succeeds or `False` if not and can take input arguments which data is automatically generated by QuickCheck.

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

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

-- reverse is distributive over append (++)
-- This test fails for explanatory reasons, for a correct
-- property xs and ys need to be swapped on the right-hand side!
reverse_distributive :: [Int] -> [Int] -> Bool
reverse_distributive xs ys = reverse (xs ++ ys) == reverse xs ++ reverse ys

-- running the tests
main :: IO ()
main = do
  quickCheck append_associative
  quickCheck reverse_reverse
  quickCheck reverse_distributive
```

When we run the tests using *main*, we get the following output:

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
*** Failed! Falsifiable (after 5 tests and 6 shrinks):
[0]
[1]
```

We see that QuickCheck generates 100 test cases for each property test and it does this by generating random data for the input arguments. We have not specified any data for our input arguments because QuickCheck is able to provide a suitable data generator through type inference. For lists and all the existing Haskell types there exist custom data generators already. We have to use a monomorphic list, in our case `Int`, and cannot use polymorphic lists because QuickCheck would not know how to generate data for a polymorphic type. Still, by appealing to genericity and polymorphism, we get the guarantee that the test case is the same for all types of a lists.

QuickCheck generates 100 test cases by default and requires all of them to pass. If there is a test case which fails, the overall property test fails and QuickCheck shrinks the input to a minimal size, which still fails and reports it as a counter example. This is the case in the last property test `reverse_distributive` which is wrong as *xs* and *ys* need to be swapped on the right-hand side. In this run, QuickCheck found a counter example to the property after 5 tests and applied 6 shrinks to find the minimal failing example of `xs = [0]` and `ys = [1]`. If we swap `xs` and `ys`, the property test passes 100 test cases just like the other two did. It is possible to configure QuickCheck to generate more or less random test cases, which can be used to increase the coverage if the sampling space is quite large - this will become useful later.

2.1.1. Generators

QuickCheck comes with a lot of data generators for existing types like `String, Int, Double, []`, but in case one wants to randomize custom data types one has to write custom data generators. There are two ways to do this. Either fix them at compile time by writing an `Arbitrary` instance or write a run-time generator running in the `Gen` Monad. The advantage of having an `Arbitrary` instance is that the custom data type can then be used as random argument to a function as in the examples above.

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

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

This implementation makes use of the `elements :: [a] → Gen a` functions, which picks a random element from a non-empty list with uniform probability. If a skewed distribution is needed, one can use the `frequency :: [(Int, Gen a)] → Gen a` function, where a frequency can be specified for each element. For example generating on average 80% `Susceptible`, 15% `Infected` and 5% `Recovered` can be achieved using this function:

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

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

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

When we have a random `Double` as input to a function but want to restrict its random range to (0,1) because it reflects a probability, we can do this easily with `newtype` and implementing an `Arbitrary` instance. The same can be done for limiting the simulation duration to a lower range than the full `Double` range.

```haskell
newtype Probability = P Double
newtype TimeRange   = T Double

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

instance Arbitrary TimeRange where
  arbitrary = T <$> choose (0, 50)
```

The simulations we run all rely on a random-number generator, thus we need a randomly initialised random-number generator each time we run a simulation. This can be easily achieved by drawing a seed from the full `Int` range and creating an `StdGen` from it:

```haskell
genStdGen :: Gen StdGen
-- min/maxBound are defined in the Haskell Prelude and
-- define the smallest and largest value of a Bounded type
genStdGen = mkStdGen <$> choose (minBound, maxBound)

instance Arbitrary StdGen where
  arbitrary = genStdGen
```

This generator then can be used to write another custom data generator which generates simulation runs. Here we give an example for the time-driven SIR:

```haskell
genTimeSIR :: [SIRState]  -- ^ Population
           -> Double      -- ^ Contact rate (beta)
           -> Double      -- ^ Infectivity (gamma)
           -> Double      -- ^ Illness duration (delta)
           -> Double      -- ^ Time Delta
           -> Double      -- ^ Time Limit
           -> Gen [(Double, (Int, Int, Int))]
genTimeSIR as beta gamma delta dt tMax
  = runTimeSIR as beta gamma delta dt tMax <$> genStdGen
```

2.1.2. Distributions

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

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

When running the test, we see the following output:

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

2.1.3. Coverage

The most powerful functions to work with test-case distributions though are `cover` and `checkCoverage`. The function `cover :: Testable prop ⇒ Double → Bool → String → prop → Property` allows to explicitly specify that a given percentage of successful test cases belong to a given class. The first argument is the expected percentage; the second argument is a `Bool` indicating whether the current test case belongs to the class or not; the third argument is a label for the coverage; the fourth argument is the property which needs to hold for the test case to succeed.

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

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

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

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

As can be seen, QuickCheck runs the default 100 test cases and prints a warning if the expected coverage is not reached. This is a useful feature but it is up to us to decide whether 100 test cases are suitable and whether we can really claim that the given coverage will be reached or not. Fortunately, QuickCheck provides the powerful function `checkCoverage :: Testable prop ⇒ prop → Property` which does this for us. When `checkCoverage` is used, QuickCheck will run an increasing number of test cases until it can decide whether the percentage in `cover` was reached or cannot be reached at all. The way QuickCheck does it, is by using sequential statistical hypothesis testing (16), thus if QuickCheck comes to the conclusion that the given percentage can or cannot be reached, it is based on a robust statistical test giving strong confidence in the result.

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

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

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

2.1.4. Emulating failure

As already mentioned, *all* test cases have to pass for the whole property test to succeed. If just a single test case fails, the whole property test fails. This requirement is sometimes too strong, especially when we are dealing with stochastic systems like ABS.

The function `cover` can be used to emulate failure of test cases and get a measure of failure. Instead of computing the `True/False` property in the last `prop` argument, we set the last argument always to `True` and compute the `True/False` property in the second `Bool` argument, indicating whether the test case belongs to the class of passed

tests or not. This has the effect that *all* test cases are successful but that we get a distribution of failed and successful ones. In combination with `checkCoverage`, this is a particularly powerful pattern for testing ABS, which allows us to test hypotheses and statistical tests on distributions as will be shown in the following chapters.

3. Conclusions

TODO

References

[1] Asta, S., Özcan, E., and Siebers, P.-O. An investigation on test driven discrete event simulation. In *Operational Research Society Simulation Workshop 2014 (SW14)* (Apr. 2014).

[2] Balci, O. Verification, Validation, and Testing. In *Handbook of Simulation*, J. Banks, Ed. John Wiley & Sons, Inc., 1998, pp. 335–393.

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

[4] Claessen, K., and Hughes, J. QuickCheck - A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 268–279.

[5] Claessen, K., and Hughes, J. Testing Monadic Code with QuickCheck. *SIGPLAN Not. 37*, 12 (Dec. 2002), 47–59.

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

[7] Epstein, J. M. Chapter 34 Remarks on the Foundations of Agent-Based Generative Social Science. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1585–1604.

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

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

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

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

[12] Ormerod, P., and Rosewell, B. Validation and Verification of Agent-Based Models in the Social Sciences. In *Epistemological Aspects of Computer Simulation in the Social Sciences*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Oct. 2006, pp. 130–140.

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

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

[15] Runciman, C., Naylor, M., and Lindblad, F. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (New York, NY, USA, 2008), Haskell '08, ACM, pp. 37–48.

[16] Wald, A. Sequential Tests of Statistical Hypotheses. In *Breakthroughs in Statistics: Foundations and Basic Theory*, S. Kotz and N. L. Johnson, Eds., Springer Series in Statistics. Springer New York, New York, NY, 1992, pp. 256–298.