# Verification & correctness of Agent-Based Simulation

JONATHAN THALER, THORSTEN ALTENKIRCH, and PEER-OLAF SIEBERS, University of Nottingham, United Kingdom

Additional Key Words and Phrases: Verification, Dependent Types, Agent-Based Simulation, Discrete Event Simulation

**ACM Reference Format:**
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Verification & correctness of Agent-Based Simulation. *Proc. ACM Program. Lang.* 9, 4, Article 39 (September 2018), 5 pages. https://doi.org/0000001.0000001

## 1 INTRODUCTION

Previous research has shown that the pure functional programming paradigm as in Haskell is very suitable to implement agent-based simulations. Building on FRP and MSFs the work developed an elegant implementation of an agent-based SIR model which was pure. By statically removing all external influences of randomness already at compile time through types, this guarantees that repeated simulation runs with the same starting conditions will always result in the same dynamics - guaranteed at compile time. This previous research focused only on establishing the basic concepts of ABS in functional programming but it did not explore the inherent strength of functional programming for verification and correctness any further than guaranteeing the reproducibility of the simulation at compile time.

This paper picks up where the previous research has left and wants to investigate the usefulness of pure and dependently typed functional programming for verification and correctness of agent-based simulation. We are especially interested if requirements of an ABS can be guaranteed on a stronger level by those paradigms, if a larger class of bugs can be excluded already at compile time and whether we can express model properties and invariants already at compile time on a type level. Further we are interested in how far we can reason about an agent-based model in a dependently typed implementation. Also we investigate the use of QuickCheck for code- and model-testing.

the key points to investigate in this report are: - testing of abs: unit- & property testing - using types for invariants / bug free - reasoning about correctness in code - reasoning about dynamics in code

Independent of the programming paradigm, there exist fundamentally two approaches implementing agent-based simulation: time- and event-driven. In the time-driven approach, the simulation is stepped in fixed $\Delta t$ and all agents are executed at each time-step - they act virtually in lock-step at the same time. The approach is inspired by the theory of continuous system dynamics (TODO: cite). In the event-driven approach, the system is advanced through events, generated by

Authors' address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk; Thorsten Altenkirch, thorsten.altenkirch@nottingham.ac.uk; Peer-Olaf Siebers, University of Nottingham, 7301 Wollaton Rd, Nottingham, NG8 1BB, United Kingdom, peer-olaf.siebers@nottingham.ac.uk.


2475-1421/2018/9-ART39 $15.00
https://doi.org/0000001.0000001

the agents, and the global system state changes by jumping from event to event, where the state is held constant in between. The approach is inspired by discrete event simulation (DES) (TODO: citation) which is formalized in the DEVS formalism [**?** ].

In a preceding paper we investigated how to derive a time-driven pure functional ABS approach in Haskell (TODO: cite my paper). We came to quite satisfactory results and implemented also a number of agent-based models of various complexity (TODO: cite schelling, sugarscape, agent zero). Still we identified weaknesses due to the underlying functional reactive programming (FRP) approach. It is possible to define partial implementations which diverge during runtime, which may be difficult to determine for complex models for a programmer at compile time. Also sampling the system with fixed $\Delta t$ can lead to severe performance problems when small $\Delta t$ are required, as was shown in our paper. The later problem is well known in the simulation community and thus as a remedy an event-driven approach was suggested [**?** ]. In this paper for the first time, we derive a pure functional event-driven agent-based simulation. Instead of using Haskell, which provides already libraries for DES [**?** ], we focus on the dependently typed pure functional programming language Idris. In our previous paper we hypothesised that dependent types may offer interesting new insights and approaches to ABS but it was unclear how exactly we can make use of them, which was left for further research. In this paper we hypothesise that, as opposed to a time-driven approach, the even-driven approach is especially suited to make proper use of dependent types due to its different nature. Note that both a pure functional event-driven approach to ABS *and* the use of dependent types in ABS has so far never been investigated, which is the unique contribution of this paper. If we can construct a dependently typed program of the SIR ABM which is total, then we have a proof-by-construction that the SIR model reaches a steady-state after finite time

Dependent Types are the holy grail in functional programming as they allow to express even stronger guarantees about the correctness of programs and go as far where programs and types become constructive proofs [**?** ] which must be total by definition [**?** ], [**?** ], [**?** ], [**?** ]. Thus the next obvious step is to apply them to our pure functional approach of agent-based simulation. So far no research in applying dependent types to agent-based simulation exists at all and it is not clear whether dependent types do make sense in this setting. We explore this for the first time and ask more specifically how we can add dependent types to our pure functional approach, which conceptual implications this has for ABS and what we gain from doing so. Note that we can only scratch the surface and lay down basic ideas and leave a proper in-depth treatment of this topic for further research. We use Idris [**?** ], [**?** ] as language of choice as it is very close to Haskell with focus on real-world application and running programs as opposed to other languages with dependent types e.g. Agda and Coq which serve primarily as proof assistants.

Dependent Types promise the following:

(1) Types as proofs - In dependently types languages, types can depend on any values and are first-class objects themselves. TODO: make more clear
(2) Totality and termination - Constructive proofs must terminate, this means a well-typed program (which is itself a proof) is always terminating which in turn means that it must consist out of total functions. A total function is defined by [**?** ] as: it terminates with a well-typed result or produces a non-empty finite prefix of a well-typed infinite result in finite time. Idris is turing complete but is able to check the totality of a function under some circumstances but not in general as it would imply that it can solve the halting problem. Other dependently typed languages like Agda or Coq restrict recursion to ensure totality of all their functions - this makes them non turing complete.

Validation & Verification in ABS http://www2.econ.iastate.edu/tesfatsi/VVAccreditationSimModels.OBalci1998.pdf : verification = are we building the model right? validation = are we building the right model?

good paper http://www2.econ.iastate.edu/tesfatsi/VVAccreditationSimModels.OBalci1998.pdf : very nice 15 guidelines and life cycles, VERY valuable for background and introduction

http://www2.econ.iastate.edu/tesfatsi/VVSimulationModels.JKleijnen1995.pdf : suggests good programming practice which is extremely important for high quality code and reduces bugs but real world practice and experience shows 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 huge 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. this is especially easy in functional programming as composability is much better than with traditional oop 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. Also the paper suggests using animations to visualise the processes within the simulation for verification purposes (of course they note that animation may be misleading when one focuses on too short simulation runs).

good paper:https://link.springer.com/chapter/10.1007/978-3-642-01109-2_10 -> verification. "This is essentially the question: does the model do what we think it is supposed to do? Whenever a model has an analytical solution, a condition which embraces almost all conventional economic theory, verification is a matter of checking the mathematics." -> validation: "In an important sense, the current process of building ABMs is a discovery process, of discovering the types of behavioural rules for agents which appear to be consistent with phenomena we observe." => can we encode phenomena we observe in the types? can we use types for the discovery process as well? can dependent types guide our exploratory approach to ABS? -> "Because such models are based on simulation, the lack of an analytical solution (in general) means that verification is harder, since there is no single result the model must match. Moreover, testing the range of model outcomes provides a test only in respect to a prior judgment on the plausibility of the potential range of outcomes. In this sense, verification blends into validation."

either one has an analytical model as the basis of an agent-based model (ABM) or one does not. In the former case, e.g. the SIR model, one can very easily validate the dynamcis generated by the ABM to the one generated by the analytical solution (e.g. through System Dynamics). Of course the dynamics wont be exactly the same as ABS discretisizes the approach and introduces stochastics which means, one must validate averaged dynamics. In the latter case one has basically no idea or description of the emergent behaviour of the system prior to its execution. 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. (eventuell hilft hier hans vollbrecht weiter: Simulation hat hier den Sinn, die Controller anhand der Roboteraufgabe zu validieren, Bei solchen Simulationen ist man interessiert an allen mÃűglichen Sequenzen, und da das meist zu viele sind, an einer mÃűglichst gut verteilten Stichprobenmenge. Hier geht es weniger um richtige Zeitmodellierung, sondern um den Test aller mÃűglichen Ereignissequenzen.)

look into DEVS

TODO: the implementation phase is just one stage in a longer process http://jasss.soc.surrey.ac.uk/12/1/1.html

WE FOCUS ON VERIFICATION important: we are not concerned here with validating a model with the real world system it simulates. this is an entirely different problem and focuses on the questions if we have built the right model. we are interested here in extremely strong verification: have we built the model right? we are especially interested in to which extend purely and dependently-typed functional programming can support us in this task.

http://jasss.soc.surrey.ac.uk/8/1/5.html: "For some time now, Agent Based Modelling has been used to simulate and explore complex systems, which have proved intractable to other modelling approaches such as mathematical modelling. More generally, computer modelling offers a greater flexibility and scope to represent phenomena that do not naturally translate into an analytical framework. Agent Based Models however, by their very nature, require more rigorous programming standards than other computer simulations. This is because researchers are cued to expect the unexpected in the output of their simulations: they are looking for the 'surprise' that shows an interesting emergent effect in the complex system. It is important, then, to be absolutely clear that the model running in the computer is behaving exactly as specified in the design. It is very easy, in the several thousand lines of code that are involved in programming an Agent Based Model, for bugs to creep in. Unlike mathematical models, where the derivations are open to scrutiny in the publication of the work, the code used for an Agent Based Model is not checked as part of the peer-review process, and there may even be Intellectual Property Rights issues with providing the source code in an accompanying web page."

http://jasss.soc.surrey.ac.uk/12/1/1.html: "a prerequisite to understanding a simulation is to make sure that there is no significant disparity between what we think the computer code is doing and what is actually doing. One could be tempted to think that, given that the code has been programmed by someone, surely there is always at least one person - the programmer - who knows precisely what the code does. Unfortunately, the truth tends to be quite different, as the leading figures in the field report, including the following: You should assume that, no matter how carefully you have designed and built your simulation, it will contain bugs (code that does something different to what you wanted and expected), "Achieving internal validity is harder than it might seem. The problem is knowing whether an unexpected result is a reflection of a mistake in the programming, or a surprising consequence of the model itself. [âĂę] As is often the case, confirming that the model was correctly programmed was substantially more work than programming the model in the first place. This problem is particularly acute in the case of agent-based simulation. The complex and exploratory nature of most agent-based models implies that, before running a model, there is some uncertainty about what the model will produce. Not knowing a priori what to expect makes it difficult to discern whether an unexpected outcome has been generated as a legitimate result of the assumptions embedded in the model or, on the contrary, it is due to an error or an artefact created in the model design, its implementation, or its execution."

general requirements to ABS - only rely on past -> solved with Arrowized FRP - bugs due to implicitly mutable state -> can be ensured by pure functional programming - ruling out external sources of non-determinism / randomness -> can be ensured by pure functional programming - correct interaction protocols -> can be ensured by dependent state machines - deterministic time-delta -> TODO: can we ensure it through dependent-types at type-level? - repeated runs lead to same dynamics -> can be ensured by pure functional programming

dependent-types: -> encode model-invariants on a meta-level -> encode dynamics (what? feedbacks? positive/negative) on a meta-level -> totality equals steady-state of a simulation, can enforce totality if required through type-level programming

## 2 TESTING

TODO: explore ABS testing in pure functional Haskell - testing of distributions e.g. all agents recover on average after illness duration - can we express model properties in tests e.g. quickcheck? - isolated tests: how easy can we test parts of an agent / simulation?

## 3 DEPENDENTLY TYPED SIR

Intuitively, based upon our model and the equations we can argue that the SIR model enters a steady state as soon as there are no more infected agents. Thus we can informally argue that a SIR model must always terminate as:

(1) Only infected agents can infect susceptible agents.
(2) Eventually after a finite time every infected agent will recover.
(3) There is no way to move from the consuming *recovered* state back into the *infected* state [1].

Thus a SIR model must enter a steady state after finite steps / in finite time.

This result gives us the confidence, that the agent-based approach will terminate, given it is really a correct implementation of the SD model. Still this does not proof that the agent-based approach itself will terminate and so far no proof of the totality of it was given. Dependent Types and Idris ability for totality and termination checking should theoretically allow us to proof that an agent-based SIR implementation terminates after finite time: if an implementation of the agent-based SIR model in Idris is total it is a proof by construction. Note that such an implementation should not run for a limited virtual time but run unrestricted of the time and the simulation should terminate as soon as there are no more infected agents. We hypothesize that it should be possible due to the nature of the state transitions where there are no cycles and that all infected agents will eventually reach the recovered state. Abandoning the FRP approach and starting fresh, the question is how we implement a *total* agent-based SIR model in Idris. Note that in the SIR model an agent is in the end just a state-machine thus the model consists of communicating / interacting state-machines. In the book [**?** ] the author discusses using dependent types for implementing type-safe state-machines, so we investigate if and how we can apply this to our model. We face the following questions: how can we be total? can we even be total when drawing random-numbers? Also a fundamental question we need to solve then is how we represent time: can we get both the time-semantics of the FRP approach of Haskell AND the type-dependent expressivity or will there be a trade-off between the two?

– TODO: express in the types – SUSCEPTIBLE: MAY become infected when making contact with another agent – INFECTED: WILL recover after a finite number of time-steps – RECOVERED: STAYS recovered all the time

– SIMULATION: advanced in steps, time represented as Nat, as real numbers are not constructive and we want to be total – terminates when there are no more INFECTED agents

Received March 2018; revised March 2018; accepted March 2018

[1] There exists an extended SIR model, called SIRS which adds a cycle to the state-machine by introducing a transition from recovered to susceptible but we don't consider that here.